CCured in the Real World

Jeremy Condit Matthew Harren Scott McPeak George C. Necula Westley Weimer
University of California, Berkeley
{jcondit,matth,smcpeak,necula,weimer}@cs.berkeley.edu *

ABSTRACT

CCured is a program transformation system that adds memory safety guarantees to C programs by verifying statically that memory errors cannot occur and by inserting run-time checks where static verification is insufficient.

This paper addresses major usability issues in a previous version of CCured, in which many type casts required the use of pointers whose representation was expensive and incompatible with precompiled libraries. We have extended the CCured type inference algorithm to recognize and verify statically a large number of type casts; this goal is achieved by using physical subtyping and pointers with run-time type information to allow parametric and subtype polymorphism. In addition, we present a new instrumentation scheme that splits CCured's metadata into a separate data structure whose shape mirrors that of the original user data. This scheme allows instrumented programs to invoke external functions directly on the program's data without the use of a wrapper function.

With these extensions we were able to use CCured on real-world security-critical network daemons and to produce instrumented versions without memory-safety vulnerabilities.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Reliability, Experimentation, Security, Languages

Keywords

Type safety, memory safety, C, run-time type information, compatibility with library code.

*This research was supported in part by the National Science Foundation Career Grant No. CCR-9875171, ITR Grants No. CCR-0085949 and No. CCR-0081588, an NSF Graduate Fellowship, 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.

PLDI'03, June 9–11, 2003, San Diego, California, USA.

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 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 simply by running the instrumented programs on inputs included with them. 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 [26]. 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.

In a previous paper [15] we described an early version of the CCured system. The fundamental concept in CCured is the *pointer qualifier*, which is used to classify each pointer into one of three categories according to how it can be used. First, pointers with the qualifier `SAFE` can be dereferenced but cannot be subject to pointer arithmetic or to type casts. They require only a null-pointer check and are as cheap to use as a reference in a type-safe language such as Java. The second qualifier, `SEQ` ("sequence"), indicates that a pointer can be used in pointer arithmetic but not in type casts. Such a pointer requires bounds checks and is instrumented to carry with it information about the bounds of the memory area to which it is supposed to point. Finally, the CCured type system has a third class of pointers with the qualifier `WILD`. In the original CCured type system, `WILD` pointers were the only ones that could be cast to other pointer types; however, they are more expensive to use because they carry type tags and require tag manipulation at run time, much like references in a dynamically typed language like Lisp. We review the CCured type system in more detail in Section 2.

The early version of CCured was already usable on existing C programs because of its ability to infer the best pointer kinds in unannotated programs. However, that system also

had some serious usability problems that made it very difficult to apply it to system software and large security-critical network daemons. In this paper we describe extensions that make it possible to apply CCured to these programs, and we describe our experience in doing so.

The major usability problem in the original version of CCured was due to incompatibilities between the CCured representation of types and the standard C representation used by precompiled libraries. One notable example is the multi-word representation of the CCured `SEQ` and `WILD` pointers. Also, objects referenced by `WILD` pointers must contain tags used to perform CCured's run-time checks. 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.

In this paper we describe a three-point solution to alleviating the incompatibility problem.

First, we observe that in the presence of structures and unions, most of the type casts can 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 extend 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 have 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 4.1, we describe our technique for automatically instantiating user-specified wrappers in many contexts.

Third and 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 represented in a separate data structure whose shape mirrors that of the program data itself. Since this separation would incur a performance penalty if used for all data, the CCured inference algorithm has been extended to limit the use of this new representation to only those types where it is required in order to preserve both soundness and compatibility. This mechanism is described in more detail in Section 4.2.

The new capabilities of CCured have enabled us to apply it to a realistic sampling of security-critical applications. 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 in using CCured for these applications in Section 5. Perhaps if these instrumented binaries see 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.

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

Figure 1: CCured representations. Given a CCured type τ (with pointer qualifiers), $\mathsf{Rep}(\tau)$ gives its layout and representation.

2. CCURED TYPES AND CHECKS

In this section we review in more detail the typing rules and the run-time checks used in the original CCured implementation [15]. The CCured type system can be viewed as two universes that coexist soundly. On 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 untyped pointers, for which we cannot count on the static type; instead, we rely on run-time tags to differentiate pointers from non-pointers.

CCured provides three pointer kinds with varying capabilities and costs. 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.

The original CCured required equal types on both sides of an assignment statement involving `SAFE` or `SEQ` pointers. Type casts between unequal pointer types are called *bad casts*, and we consider the pointers involved to be untyped, 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.

Since CCured pointers must carry bounds and 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 1. We refer to the pointer, base, and end parts of a sequence pointer `x` as `x.p`, `x.b`, and `x.e`, respectively. We store the "base" and "end" fields, which contain *metadata* information, with the pointer itself, thus turning C's one-word pointer representation into a multi-word structure. An area referenced by `WILD` pointers contains additional metadata that indicates the size of the area and whether each word in the area is a pointer.

Reads and writes through CCured pointers require run-time checks. Reads through `SAFE` pointers require merely a `null` check, whereas reads through `SEQ` pointers require a check that the pointer is in bounds (i.e., $\texttt{x.b} \leq \texttt{x.p} \leq \texttt{x.e} - \texttt{sizeof}(\tau)$, where τ is the type being read). Reading through a `WILD` pointer requires bounds checks similar to the checks performed on `SEQ` pointers, although the upper bound for a `WILD` pointer is stored in the memory area itself. In addition, `WILD` pointer reads require run-time tag checks to ensure that the program never reads a pointer from a location that contains an integer. Memory writes perform the same checks as memory reads, but writes must also verify that a stack pointer is not being stored in the heap and, for `WILD` pointers, they must update the tags.

2.1 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, with the address of every variable, and with the address of every structure field. The inference algorithm then assigns a pointer kind to every qualifier variable such that the resulting program type checks in the CCured system.

The CCured inference finds the best kind for a pointer and assigns `SAFE` and `SEQ` qualifiers whenever possible. In CCured's algorithm, every occurrence of pointer arithmetic produces a constraint that the pointer involved must be `SEQ` or `WILD`. Also, every occurrence of a bad cast produces a constraint that the pointers involved must be `WILD`. Pointers without any such constraints are inferred `SAFE`.

To ensure soundness in CCured, we cannot allow both an untyped and a statically-typed pointer to refer to the same location, for otherwise the untyped pointer could write arbitrary data in the referenced location and invalidate the assumptions of its statically-typed alias. The other soundness condition is that we cannot allow an untyped pointer to point to a statically-typed pointer, or else we could alter the value of the statically-typed pointer in a way that violates its invariant. These conditions mean that if a pointer qualifier becomes `WILD` then all qualifiers in its base type become `WILD` as well. Similarly, the qualifiers of pointers that are assigned to or from a `WILD` pointer become `WILD` as well.

This linear-time inference algorithm, restricted to the case of a language without structured types, is described in detail and proved to be both correct and optimal in a previous paper [15]. However, the trivial extension of the algorithm to structured types has some serious limitations. In particular, there are many casts between pointers to structured types that our algorithm considers bad. In the next section, we describe two new extensions to the CCured type system that we have found to be very effective in reducing the number of bad casts.

3. COPING WITH CASTS

Many C programs make heavy use of casts between pointer types. In the original CCured, all pointer qualifiers involved in non-trivial casts must become `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 following code fragment, which contains object-oriented style subtype polymorphism.

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

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

Siff et al. [22] observe that a large fraction of the casts between unequal types in real programs are either upcasts or downcasts, and our experiments support this observation. In particular, we have observed that around 63% of casts are between identical types. The remaining 37% were bad casts in the original CCured. Of these bad casts, about 93% are safe upcasts and 6% are downcasts. Less than 1% of all casts fall outside of these categories, and must still be considered bad even in the presence of mechanisms that handle downcast and upcasts. 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 [6, 22].

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

$$\tau \lesssim \tau' \stackrel{\text{def}}{\iff} \exists \tau''.\ \tau \approx \texttt{struct}\{\tau'; \tau''; \}$$

Physical type equality $\approx$ is defined as the smallest equivalence relation generated by the following equations:

$$\begin{aligned}
\tau * \texttt{WILD} &\approx \tau' * \texttt{WILD} \\
\tau[1] &\approx \tau \\
\tau[n_1 + n_2] &\approx \texttt{struct}\{\tau[n_1]; \tau[n_2]; \} \\
\texttt{struct}\{\tau_1; \texttt{void}; \} &\approx \tau_1 \\
\texttt{struct}\{\tau_1; \texttt{struct}\{\tau_2; \tau_3; \}; \} &\approx \texttt{struct}\{\texttt{struct}\{\tau_1; \tau_2; \}; \tau_3; \}
\end{aligned}$$

Note that care must be taken to account for structure padding when using the structure associativity rule; we omit

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
$(\tau' * \texttt{RTTI})x$	$x : \tau * \texttt{RTTI},\ \tau' \lesssim \tau$	$\texttt{assert}(\texttt{isSubtype}(x.t, \texttt{rttiOf}(\tau'));\ x$
$(\tau' * \texttt{SAFE})x$	$x : \tau * \texttt{RTTI}$	$\texttt{assert}(\texttt{isSubtype}(x.t, \texttt{rttiOf}(\tau'));\ x.p$

Figure 2: 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.

the details. With these definitions the more relaxed CCured typing rules for casts are the following:

$$\frac{e : \tau' * \texttt{SAFE} \quad \tau' \lesssim \tau}{(\tau * \texttt{SAFE})e : \tau * \texttt{SAFE}} \qquad \frac{e : \tau' * \texttt{SEQ} \quad \exists n, n'.\ \tau'[n'] \approx \tau[n]}{(\tau * \texttt{SEQ})e : \tau * \texttt{SEQ}}$$

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 [6, 22], void* is allowed in the smaller aggregate in any position where a regular pointer is present in the larger one. This approach is unsound; instead, void should be considered to be the empty structure 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 multi-dimensional arrays (in which case one of the n or n' is typically 1) and is necessary for handling many non-trivial C programs.

Changes to the Inference Algorithm. In order to handle structures and physical subtyping, the original CCured inference algorithm [15] must be extended to pay special attention to casts. If there is a cast from $\tau_1 *$ to $\tau_2 *$, we examine τ_1 and τ_2 in tandem using the physical type equality rules, and we emit pointer kind constraints to make sure that τ_1 is a subtype of τ_2. Basically, if τ_2's representation is a prefix of τ_1's representation, the qualifiers can remain SAFE. Otherwise, they must be WILD. For reasons of space we omit many details here; a complete discussion of inference is available as a technical report [27].

3.2 Downcasts and Run-Time Type Info.

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 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 $\tau *$ 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 $\tau *$ pointer if needed; alternatively, it can be cast to some physical subtype of τ with a run-time check.

In Figure 2 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.

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 ($\tau \lesssim \tau'$) and the source type has subtypes, then the RTTI pointer kind propagates against the data flow: $q' = \texttt{RTTI} \wedge (\exists \tau''.\ \tau'' \lesssim \tau) \implies q = \texttt{RTTI}$.
- 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. COMPATIBILITY WITH LIBRARIES

It is often necessary to link cured programs with 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.

```
#pragma ccuredWrapperOf("strchr_wrapper", "strchr")
char* strchr_wrapper(char* str, int chr) {
  __verify_nul(str);   // check for NUL termination
  // call underlying function, stripping metadata
  char *result = strchr(__ptrof(str), chr);
  // build a wide CCured ptr for the return value
  return __mkptr(result, str);
}
```

Figure 3: A wrapper for strchr.

4.1 Library Wrappers

One approach to the problem of library compatibility is to write wrapper functions for external library functions. CCured has been structured so that a program that tries to communicate with a non-CCured library using multi-word pointers will fail to link rather than crash at run time. To link correctly with a function that is not instrumented, CCured must:

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 pack or unpack multi-word pointers.

We accomplish these tasks by requiring the programmer to provide a small wrapper specification for each external function called by a program.[1] For example, Figure 3 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 all calls to strchr with the body of this wrapper, in which the helper functions __verify_nul, __ptrof, and __mkptr are replaced with specialized code depending on the pointer kinds of their arguments and results. A single wrapper function works with any set of inferred qualifiers. Notice that the wrapper specification can also include checks of preconditions of the library functions.

We have implemented wrappers for about 100 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.

4.2 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;
};
```

[1] In practice, this specification is only necessary for functions whose actual arguments contain multi-word pointers.

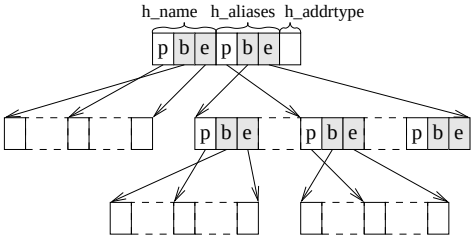

Figure 4: Interleaved representation for `struct hostent`. Array-bounds metadata (gray) is interspersed with data (white).

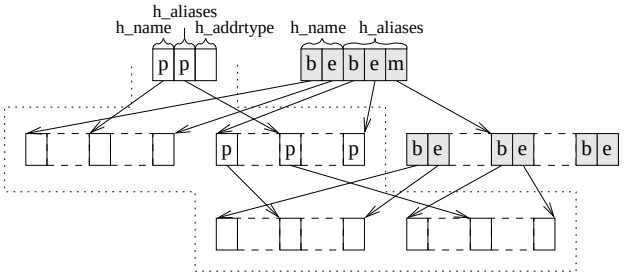

Figure 5: Non-interleaved representation for `struct hostent`. Metadata (gray) has been separated into a parallel data structure so that the data (white, boxed) has the format expected by the C library. The `m` field is a pointer to the array's metadata.

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 4. 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 5 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

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

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

Figure 6: 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.

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, 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 1.

Formal definitions for the functions C and Meta are given in Figure 6. 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. Note that we currently do not support this compatible representation for `WILD` pointers.

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

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

Figure 7: The Rep function can be extended to handle NOSPLIT types that contain SPLIT types. This definition extends the definition given in Figure 1, which considers only NOSPLIT types.

$\mathsf{Meta}(\tau)$. Correspondingly, a value of type τ NOSPLIT is represented using the type $\mathsf{Rep}(\tau)$, which contains interleaved data and metadata.

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 7, 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 5. 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 5. 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.

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.

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.

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.

5. EXPERIMENTS

We tested our system 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 runtime checks is dwarfed by the costs of input/output operations. Our experiments allowed us to detect a number of bugs in existing programs and enable us to run safety-critical code without fear of memory-based security errors such as buffer overruns.

We first tested CCured with the `Spec95` [24], `Olden` [3], and `Ptrdist-1.1` [2] 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 resulted in modifications to about 1 in 100 lines of source code. In the process we discovered a number of bugs in these benchmarks, including several array bounds violations and a `printf` that is passed a `FILE*` when expecting a `char*` [15].

CCured's safety checks added between 7 and 56% to the running times of these tests. For comparison, we also tried these tests with Purify version 2001A [8], which increased running times by factors of 25–100. Purify modifies C binaries directly to detect memory leaks and access violations by keeping two 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 [11], a property that Fischer and Patil [17] show to be important. We commented on some of these experiments in a previous paper [15].

We also ran comparisons against Valgrind [20], 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 9. Both Purify and Valgrind miss many memory errors that CCured catches; in particular, these other tools do not catch out-of-bounds array indexing on stack-allocated arrays.

For the remainder of this section, we focus on experiments that we were able to perform only after extending CCured as explained in this paper.

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 4.1.

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 [19]. 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.

Module Name	Lines of code	% sf/sq/w/rt	**CCured Ratio**
`asis`	149	72/28/0/0	**0.96**
`expires`	525	77/23/0/0	**1.00**
`gzip`	11648	85/15/0/0	**0.94**
`headers`	281	90/10/0/0	**1.00**
`info`	786	86/14/0/0	**1.00**
`layout`	309	82/18/0/0	**1.01**
`random`	131	85/15/0/0	**0.94**
`urlcount`	702	87/13/0/0	**1.02**
`usertrack`	409	81/19/0/0	**1.00**
`WebStone`	n/a	n/a	**1.04**

Figure 8: Apache Module Performance. The "sf/sq/w/rt" column show the percentage of (static) pointer declarations which were inferred `SAFE`, `SEQ`, `WILD` and `RTTI`, respectively. A ratio of 1.04 means that the CCured module was 4% slower than the original.

Figure 8 shows the performance of these modules on tests consisting of 1,000 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.

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 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 perform 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. Porting `sbull` to CCured involved changing about 20 lines of code, and `pcnet32` required only 5 changes.

The performance measurements are shown in Figure 9. `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 (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 9 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.

Run-time Type Information. In one of the first uses of the new `RTTI` pointer kind, we revisited a previous experiment. 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

Name	Lines of code	% sf/sq/w/rt	**CCured Ratio**	Valgrind Ratio
`pcnet32`	1661	92/8/0/0	**0.99**	
ping			**1.00**	
`sbull`	1013	85/15/0/0	**1.00**	
seeks			**1.03**	
`ftpd`	6553	79/12/9/0	**1.01**	9.42
`OpenSSL`	177426	67/27/0/6	**1.40**	42.9
cast			**1.87**	48.7
bn			**1.01**	72.0
`OpenSSH`	65250	70/28/0/3		
client			**1.22**	22.1
server			**1.15**	
`sendmail`	105432	65/34/0/1	**1.46**	122
`bind`	336660	79/21/0/0	**1.81**	129
tasks			**1.11**	81.4
sockaddr			**1.50**	110

Figure 9: System software performance. A ratio of 1.03 means the CCured version is 3% slower than the original. Not all tests were applicable to Valgrind.

with only 1% of the pointers becoming `RTTI` instead. This result shows how far the `WILD` qualifier can spread from bad casts. Overall, the slowdown is reduced to 45%.

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 match 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.[2] These changes allowed us to compile `OpenSSL` with only two "trusted" casts, which were needed for pseudorandom 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 rulestrings. 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 9 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 is 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 use 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 are using 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 [5]. 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 were not type safe: `union`s became `struct`s, 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 2 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 9 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 530 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 turn on the use of `RTTI`, 150 of the bad casts (28%) proved to be downcasts that can be checked at run time. We instructed CCured to trust the remaining 380 bad casts rather than use `WILD` pointers, therefore 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 380 casts.

Figure 9 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%.

Compatible Pointer Representations. 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 4.2 determined that 6% of the pointers in the program should have split types and that 31% of these pointers need a metadata pointer. The large number of metadata pointers is 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

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

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.

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 is 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 many security holes can be prevented. Finally, we have detected several bugs in the programs we tested.

6. RELATED WORK

There have been many attempts to design C-like languages or language subsets that are provably type safe. The Cyclone language [10] is expressive, gives programmers a high degree of control, and has been used on similar types of programs (e.g., device drivers). Smith et al. [23] present a type-safe polymorphic dialect of C that includes most of C's features (and higher-order functions, which our current system handles weakly) but lacks casts and structures. However, these approaches work only for programs written in the given dialect.

Ramalingam et al. [18] have presented an algorithm for finding the coarsest acceptable type for structures in C programs. Chandra and Reps [6] present a method for physical type checking of C programs based on structure layout in the presence of casts. Siff et al. [22] report that many casts in C programs are safe upcasts and present 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).

Another common approach is to add run-time checks to C programs. Kaufer et al. [12] present 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 runs about 200 times slower than normal. Austin et al. [2] store extra information with each pointer and achieve safety at the cost of a large overhead (up to 5 times slower) and a lack of library compatibility. Jones and Kelly [11] store extra information for run-time checks in a splay tree, allowing safe code to work with unsafe libraries. This approach results in a slowdown factor of 5 to 6. Fischer and Patil have presented a system that uses a second processor to perform the bounds checks [16]. Loginov et al. [14] store type information with each memory location, incurring a slowdown factor of 5 to 158. This extra information allows them to perform more detailed checks than CCured can, and they can detect when stored types do not match declared types or when union members are accessed out of order. The approaches of Austin et al. and Jones and Kelly are comparable to the implementation of CCured's `WILD` pointers. However, beyond array bounds check elimination, none of these techniques use type-based static analysis to aggressively reduce the overhead of the instrumented code.

The global splay tree used by Jones and Kelly [11] 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 [16] maintain 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.

The pointer kind qualifiers used in CCured are a special case of type qualifiers [7]. The CCured inference algorithm bears some resemblance to Henglein's inference algorithm [9], but we also consider physical subtyping, pointer arithmetic, updates and multiple pointer kinds. Henglein's algorithm has the nice feature that it does not require any type information to be present in the program. However, we believe that his algorithm does not extend to the more complex language we consider here and also that existing C types contain valuable information that should be used to make inference both simpler and more predictable.

An entire body of research [4, 9, 13, 21, 25, 28] examines the notion of a Dynamic type whose values are $\langle \text{type}, \text{ptr} \rangle$ packages. Such a value can only be used by first extracting and checking the type. In particular, one can only write val-

ues 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. Abadi et al. [1] study the theoretical aspects of adding a Dynamic type to the simply-typed λ-calculus and discuss extensions to polymorphism and abstract data types. CCured's `RTTI` qualifier is similar, but we combine it with an inference algorithm based on physical subtyping.

7. 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 paper 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 incorporate 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 describe 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 remaining compatible 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 paper, 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, and the anonymous reviewers for their helpful comments on this paper, and the members of the Open Source Quality group for their advice and assistance throughout the project.

8. REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN Notices*, 29(6):290–301, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[3] M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines.* PhD thesis, Princeton University Department of Computer Science, June 1996.

[4] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[5] CERT Coordination Center. Cert advisory ca-2003-12: Buffer overflow in sendmail. *http://www.cert.org/advisories/CA-2003-12.html*, 2003.

[6] S. Chandra and T. Reps. Physical type checking for C. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, volume 24.5 of *Software Engeneering Notes (SEN)*, pages 66–75. ACM Press, Sept. 6 1999.

[7] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, May 1–4, 1999.

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

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

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

[11] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. *AADEBUG*, 1997.

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

[13] A. Kind and H. Friedrich. A practical approach to type inference for EuLisp. *Lisp and Symbolic Computation*, 6(1/2):159–176, 1993.

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

[15] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *The 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139. ACM, Jan. 2002.

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

[17] H. Patil and C. N. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software—Practice and Experience*, 27(1):87–110, Jan. 1997.

[18] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *Symposium on Principles of Programming Languages*, pages 119–132, Jan. 1999.

[19] SecuriTeam.com. PHP3 / PHP4 format string vulnerability. *http://www.securiteam.com/securitynews/6O00T0K03O.html*, Dec. 2000.

[20] J. Seward. Valgrind, an open-source memory debugger for x86-GNU/Linux. Technical report, *http://developer.kde.org/~ sewardj/*, 2003.

[21] M. Shields, T. Sheard, and S. L. P. Jones. Dynamic typing as staged type inference. In *Symposium on Principles of Programming Languages*, pages 289–302, 1998.

[22] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *1999 ACM Foundations on Software Engineering Conference (LNCS 1687)*, volume 1687 of *Lecture Notes in Computer Science*, pages 180–198. Springer-Verlag / ACM Press, September 1999.

[23] G. Smith and D. Volpano. A sound polymorphic type system for a dialect of C. *Science of Computer Programming*, 32(1–3):49–72, 1998.

[24] SPEC 95. Standard Performance Evaluation Corportation Benchmarks. *http://www.spec.org/osg/cpu95/CINT95*, July 1995.

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

[26] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step toward automated detection of buffer overrun vulnerabilities. In *Network Distributed Systems Security Symposium*, pages 1–15, Feb. 2000.

[27] W. Weimer. The CCured type system and type inference. Technical Report UCB-CS, University of California, Berkeley. *http://www.cs.berkeley.edu/~ weimer/TheCCuredTypeSystem.ps*, 2002.

[28] A. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 1997.

APPENDIX

A. CCURED RUN-TIME CHECKS

This appendix provides concrete details about CCured run-time invariants and checks. In order to describe the CCured typing rules and the run-time checks we shall use a simplified presentation of C. Since the array indexing operation $e_1[e_2]$ is just syntactic sugar for $*(e_1 + e_2)$, we will only consider pointer arithmetic. Comparison operations on pointer values are performed after converting the pointers to integers. We shall not describe the operations on integers since they are not interesting for memory safety.

CCured run-time invariants are shown in Figure 10. If the static type of a value is τ * `SAFE`, that value is either `null` or a valid pointer to a τ in memory. If the static type of a value is τ * `SEQ`, that value is either `null` or its `b` and `e` metadata fields delimit a valid array of τs in memory. In the `SAFE` and `SEQ` cases the type of the referent is known statically. If the static type of a value is τ * `WILD`, we must check the type at run time. We maintain the invariant that either the value is `null` or it has a valid `b` field that points to the beginning of a `WILD` object. The associated tags give the type of the last value stored in that object. As a special case of `SEQ` and `WILD` pointers, if the base field is `null` then the pointer actually represents an integer that was cast to a pointer.

The CCured typing rules and the run-time checks are shown in Figure 11. For each form of expression shown in the left column of the figure there are a number of alternative typing premises (shown in the middle column) 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 left-column expression.

In the case of memory reads we first check whether the pointer is `null`. In the case of `SEQ` and `WILD` pointers this check also verifies that the pointer was not obtained by casting from an integer. For `SEQ` and `WILD` pointers we perform a bounds check, using the notation `len(x.b)` to refer to the length of a dynamically-typed area pointed to by the base field of `x`. Finally, 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 `tag(b,p)` denotes the tag corresponding to the word pointed to by `p` inside the dynamically-typed area pointed to by `b`.

For memory writes (not shown), we perform the same checks as for reads, and additionally, we check that we do not store a stack pointer. This restriction is a conservative approximation that prevents the program from dereferencing the address of a local variable after its parent function has returned. 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 base and pointer fields to one and zero 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.

The first three lines in the "Type Casts" section of Figure 11 show that any kind of pointer can be cast to an integer, but the reverse direction prevents integers (except 0) from being cast to `SAFE` pointers. Note that even though we can disguise an integer as a `SEQ` or a `WILD` pointer, the base field will be `null`, meaning that we cannot use the pointer in a memory operation. The necessary restrictions for handling arithmetic and physical subtyping are shown in the last three lines under "Type Casts."

Finally, true pointer arithmetic is allowed only for `SEQ` and `WILD` pointers. Accessing a structure field can be viewed as a combination of casts and pointer arithmetic, however. CCured supports the creation of pointers to substructures starting from a pointer to the host structure. The representative case of $x \rightarrow f_2$ is shown. If x is a `SAFE` pointer to a structure, then it must be non-`null`; otherwise, we would obtain a `SAFE` pointer that is neither valid nor `null`. In the case of 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.

Pointer Qualifier	Representation	Invariants
$x : \tau * \texttt{SAFE}$	$\texttt{struct}\{\tau * \texttt{p};\}$	$x.p \neq \texttt{null} \implies \mathsf{IsAValid}(x.p,\ \tau)$
$x : \tau * \texttt{SEQ}$	$\texttt{struct}\{\tau * \texttt{p}, * \texttt{b}, * \texttt{e};\}$	$x.b \neq \texttt{null} \ \wedge\ x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau) \implies \mathsf{IsAValid}(x.p,\ \tau)$
$x : \tau * \texttt{WILD}$	$\texttt{struct}\{\tau * \texttt{p}, * \texttt{b};\}$	$x.b \neq \texttt{null} \ \wedge\ x.b \leq x.p \leq x.b + \mathsf{len}(x.b) - 4 \implies \mathsf{IsAValid}(x.p,\ \texttt{int})$
		$x.b \neq \texttt{null} \ \wedge\ x.b \leq x.p \leq x.b + \mathsf{len}(x.b) - 8$
		$\wedge\ \mathsf{tag}(x.b, x.p) = 1 \ \wedge\ \mathsf{tag}(x.b, x.p + 4) = 0 \implies \forall \tau'.\mathsf{IsAValid}(x.p,\ \tau' * \texttt{WILD})$

Figure 10: CCured pointer qualifier invariants. Whenever a value x of the given type exists in a well-typed CCured program, the associated invariant will hold for that value. $\mathsf{IsAValid}(x,\ \tau)$ means that x is a valid address in memory and that the last value stored there is a physical subtype of τ (see Section 3.1).

Expression	Typing Premises	Run-time checks and translation
Memory Reads		
$*x$	$x : \tau * \texttt{SAFE}$	$\mathsf{assert}(x.p \neq \texttt{null});\ *(x.p)$
$*x$	$x : \tau * \texttt{SEQ}$	$\mathsf{assert}(x.b \neq \texttt{null});\ \mathsf{assert}(x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau));\ *(x.p)$
$*x$	$x : \texttt{int} * \texttt{WILD}$	$\mathsf{assert}(x.b \neq \texttt{null});\ \mathsf{assert}(x.b \leq x.p \leq x.b + \mathsf{len}(x.b) - 4);\ *(x.p)$
$*x$	$x : \tau * \texttt{WILD} * \texttt{WILD}$	$\mathsf{assert}(x.b \neq \texttt{null});\ \mathsf{assert}(x.b \leq x.p \leq x.b + \mathsf{len}(x.b) - 8);$ $\mathsf{assert}(\mathsf{tag}(x.b, x.p) == 1);\ \mathsf{assert}(\mathsf{tag}(x.b, x.p + 4) == 0);\ *(x.p)$
Type Casts		
$(\texttt{int})x$	$x : \tau * \texttt{SAFE}$	$x.p$
$(\texttt{int})x$	$x : \tau * \texttt{SEQ}$	$x.p$
$(\texttt{int})x$	$x : \tau * \texttt{WILD}$	$x.p$
$(\tau' * \texttt{SAFE})x$	$x = 0$	$\{p = \texttt{null}\}$
$(\tau' * \texttt{SEQ})x$	$x : \texttt{int}$	$\{b = \texttt{null},\ p = x,\ e = \texttt{null}\}$
$(\tau' * \texttt{WILD})x$	$x : \texttt{int}$	$\{b = \texttt{null},\ p = x\}$
$(\tau' * \texttt{WILD})x$	$x : \tau * \texttt{WILD}$	x
$(\tau' * \texttt{SAFE})x$	$x : \tau * \texttt{SAFE},\ \tau \lesssim \tau'$	x
$(\tau' * \texttt{SEQ})x$	$x : \tau * \texttt{SEQ},\ \tau[n] \approx \tau'[n']$	x
$(\tau' * \texttt{SAFE})x$	$x : \tau * \texttt{SEQ},\ \tau[n] \lesssim \tau'$	$\mathsf{assert}(x.p = \texttt{null} \ \|\| \ x.b \neq \texttt{null});$ $\mathsf{assert}(x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau'));\ x.p$
$(\tau' * \texttt{SEQ})x$	$x : \tau * \texttt{SAFE},\ \tau'[n'] \lesssim \tau$	$\{b = x.p,\ p = x.p,\ e = x.p + \texttt{sizeof}(\tau)\}$
Miscellaneous		
$x_1 + x_2$	$x_1 : \tau * \texttt{SEQ},\ x_2 : \texttt{int}$	$\{b = x_1.b,\ p = x_1.p + x_2 * \texttt{sizeof}(\tau),\ e = x_1.e\}$
$x_1 + x_2$	$x_1 : \tau * \texttt{WILD},\ x_2 : \texttt{int}$	$\{b = x_1.b,\ p = x_1.p + x_2 * \texttt{sizeof}(\tau)\}$
$\&(x \to f_2) : \tau_2 * \texttt{SAFE}$	$x : \texttt{struct}\{\tau_1 f_1;\ \ \tau_2 f_2;\} * \texttt{SAFE}$	$\mathsf{assert}(x.p \neq \texttt{null});\ \&(x.p \to f_2)$
$\&(x \to f_2) : \tau_2 * \texttt{SAFE}$	$x : \texttt{struct}\{\tau_1 f_1;\ \ \tau_2 f_2;\} * \texttt{SEQ}$	$\mathsf{assert}(x.b \neq \texttt{null});$ $\mathsf{assert}(x.b \leq x.p \leq x.e - \texttt{sizeof}(\tau_1) - \texttt{sizeof}(\tau_2));\ \&(x.p \to f_2)$
$\&(x \to f_2) : \tau_2 * \texttt{WILD}$	$x : \texttt{struct}\{\tau_1 f_1;\ \ \tau_2 f_2;\} * \texttt{WILD}$	$\{b = x.b,\ p = \&(x.p \to f_2)\}$

Figure 11: CCured typing rules for reads, 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. For simplicity word size is assumed to be 4. All arithmetic in the right column is integer arithmetic.