Lightweight Wrappers for Interfacing with Binary Code in CCured

Matthew Harren and George C. Necula

University of California, Berkeley,
Computer Science Division,
Berkeley, CA, USA 94720-1776
{matth, necula}@cs.berkeley.edu
(510) 642-8290 fax: (510) 642-5775

Abstract. The wide use of separate compilation and precompiled libraries among programmers poses a challenge to source-code based security and analysis tools such as CCured. These tools must understand enough of the behavior of precompiled libraries that they can prevent any unsafe use of the library. The situation is even more complicated for instrumentation tools that change the layout of data to accommodate array bounds or other metadata that is necessary for safety checking.

This paper describes the solution we use with CCured: a system of context-sensitive wrapper functions. These wrappers check that library functions are invoked on valid arguments, and also maintain the extra runtime invariants imposed by CCured. We describe the design of these wrappers and our experiences using them, including the case where complex data structures are passed to or from the library.

1 Introduction

Static program analysis tools, including those that detect or prevent security problems, usually rely on the availability of source code for the programs that they analyze. For interprocedural analyses, however, the practice of linking to precompiled libraries is a large hurdle. Tools that cannot analyze binary code need a way to model the behavior of these library routines.

There are a number of reasons why an analysis tool must deal with precompiled code. The most obvious reason is that the source code for a library may be proprietary. Even for open source libraries, however, many programmers will by default install only the binaries and header files; asking users to install the source code for each library is an undesirable burden. If software security analysis tools are to become more widely used, ease-of-use is an important consideration. Finally, modeling only the essential behavior of a library component allows natural support for dynamic linking. Users may choose alternate implementations of that component without having to reanalyze the program, provided the new implementation has the same preconditions and postconditions.

This paper presents a system that enables CCured [1], a source-based security tool, to interact with library code. Throughout this paper, we use "library

K. Futatsugi et al. (Eds.): ISSS 2003, LNCS 3233, pp. 209–225, 2004.

code" to refer to any binary code compiled without CCured to which we want to link CCured code. Because CCured uses run-time instrumentation in addition to compile-time analyses, the wrappers must support the maintenance and updating of CCured's runtime metadata, in addition to modeling the pertinent aspects of the library's behavior for the static analysis. One of the primary goals of CCured is ease of use on legacy code, and a convenient mechanism for wrappers is important when we use CCured on large legacy systems.

We begin with a brief overview of the CCured system and the difficulties in using precompiled libraries with CCured. Section 3 describes the wrapper functions we use to solve this problem. We discuss in Section 4 how these wrappers make use of the context-sensitivity feature of CCured.

Section 5 extends these wrappers to more complicated data structures. We provide examples of our wrappers in Section 6, and discuss related work in Section 7.

2 CCured

CCured is an interprocedural, flow-insensitive analysis and transformation tool that guarantees type and memory safety in C programs. By classifying pointers into "kinds" based on their usage, CCured is able to insert the necessary runtime checks to ensure memory safety. These runtime checks require that *metadata* be stored with many pointers to provide, for example, the bounds of the array being manipulated, or the dynamic type of an object.

In this paper, we will consider only the three CCured pointer kinds shown in Figure 1.

- A `SAFE` pointer is an ordinary, one-word C pointer that cannot be used with pointer arithmetic. `SAFE` pointers, if not null, point to valid memory locations of the appropriate type.
- A `SEQ` (sequence) pointer has a three-word representation: the pointer plus the bounds on the *home area* for this array. The pointer may be incremented or decremented, but the bounds do not change unless the pointer is overwritten entirely. `SEQ` pointers use the following invariant: if the pointer is within the bounds it specifies, then dereferencing it will yield an object of the correct type.
- An `FSEQ` (forward sequence) pointer is an optimization for a common case of sequence pointers: when no lower-bound check is needed (because the pointer is never decremented), we need only a two-word representation. `FSEQ` pointers maintain the invariant that if the pointer value is less than the specified upper bound, then dereferencing it will yield an object of the correct type.

We prefer `SAFE` pointers to `FSEQ`, and `FSEQ` to `SEQ`, for performance reasons: `SAFE` pointers use less memory and require fewer checks at each access. Note that a `SEQ` pointer can be easily coerced to an `FSEQ`, and an `FSEQ` to a `SAFE`, by dropping unneeded metadata and performing the runtime checks.

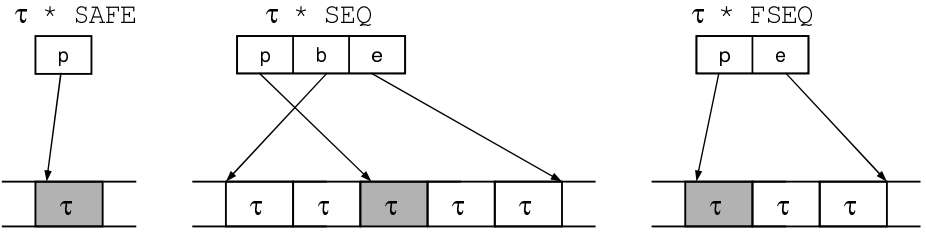

Fig. 1. `SAFE`, `SEQ` and `FSEQ` pointers, respectively, to the shaded memory locations

2.1 An Example

In the following code fragment,

```
1    char* a = (char*)malloc(mysize);
2    char* b = a;
3    b[5] = '.';
4    char* c = b;
5    char* d = c + 5;
```

CCured infers that variable b must carry an upper bound so that the array access on line 3 can be checked for safety at runtime. This requirement is transitively applied to variable a due to the assignment at line 2, since b must get its upper-bound from a. Similarly, variable c requires an upper bound because of the arithmetic on line 5, and this constraint also flows to b and a. We choose the best assignment of kinds that satisfies these constraints, and make variables a,b and c `FSEQ`. Variable d needs no bounds checks, so we make it `SAFE`.

Inferring these constraints requires a whole-program analysis. If lines 2 and 3 above are enclosed in a separate function "`setFifth`" (as shown below) we will need to pass the constraint that b requires an upper bound across the function boundary.

```
1    char* a = (char*)malloc(mysize);
2
3    char* c = setFifth(a);
4
5    char* d = c + 5;
  ...
6  char* setFifth(char* b) {
7    b[5] = '.';
8    return b;
9  }
```

Suppose `setFifth` is a precompiled library function, and its code is not visible to CCured. CCured will not know that it has to ensure that the argument b can be used in a memory operation. One solution is to write a wrapper function that understands the semantics of `setFifth` and will check that its argument has the

right length. Because the library uses C-style pointers, the wrapper also needs to pass a one-word pointer to `setFifth`, and convert the return value back to a CCured pointer, as shown below:

```
1    char* a = (char*)malloc(mysize);
2
3    char* c = setFifth_wrapper(a);
4
5    char* d = c+5;
  ...
6 char* setFifth(char* b); //Defined in a library.
7
8 char* setFifth_wrapper(char* b) {
9    if (LENGTH_OF(b) <= 5) { Error, abort the program ... }
10   //Call the real function:
11   char* retval = setFifth(CONVERT_TO_C(b));
12   return CONVERT_FROM_C(retVal);
13 }
```

However, this wrapper would need to be written in a kind-polymorphic way — the pointer kinds that a function uses may change depending how the function is used. For example, if line 5 in the example above were changed to

```
5    char* d = c - 10;
```

then variable c would need a lower bound. This requirement would force the return type of `setFifth_wrapper` to be a `SEQ` pointer, which in turn would force the argument to the wrapper to be `SEQ` so that we have a lower bound on b available for the operation labeled *CONVERT_FROM_C*. We would like to do this polymorphically, so that one such use of `setFifth_wrapper` does not force all uses of `setFifth_wrapper` to pass a `SEQ` value for b.

Our solution to the problem of library code in CCured is a system that makes writing these wrappers easy, and avoids the need to write separate wrappers for different pointer kinds. The wrappers we describe in this paper will:

1. Describe what constraints, such as buffer lengths, the external function requires on its inputs, and
2. Perform appropriate runtime actions to check these constraints, to convert CCured pointers to C, and to convert C pointers to CCured.

Although CCured cannot guarantee that the library functions are memory safe, the wrappers can guarantee that the function's preconditions are met, which will go a long way towards ensuring correct behavior.

3 Simple Wrappers for CCured

We present a mechanism for specifying wrapper functions that accomplishes both of the above points. At compile time, we generate constraints about what

metadata should be carried with each pointer, and at runtime the specification is treated as a function that manipulates metadata, performs checks, and calls the underlying library code.

The wrappers are written as ordinary C code using a number of helper functions listed in Table 1. We can divide these helpers into two main groups:

- **Those That Operate on Wide Pointers.** (Group A in Table 1) Helper functions such as `__ptrof` ("pointer of") and `__ensure_length` read the metadata maintained in CCured's wide pointers and perform necessary checks. These functions also extract the data from a wide pointer for passing to the library.
- **Those That Operate on Standard Pointers.** (Group B in Table 1) Helper functions such as `__mkptr` ("make pointer") build wide pointers that CCured can use from the standard pointers returned by a library call.

Except for `__check_string`, which relies on a complete scan of the buffer, the operations shown here are constant-time functions requiring only a few simple instructions.

We use the type "`void *`" in Table 1 to denote helper functions that can be used on pointers of any type, but CCured's typechecking of helper functions is not as lax as these function signatures might imply. In fact, CCured will guarantee at compile-time that the arguments and return values at each call to a helper function have the same type. This prevents programmers from accidentally changing the apparent type of a pointer when passing it through a helper function, and it also allows CCured to maintain necessary invariants regarding the pointer base types that we do not discuss in this paper.

Figure 2 shows a wrapper specification for `crypt`, a function that returns a cryptographic hash of the `key` argument and the first two characters of the `salt` argument. The "`#pragma ccuredwrapper`" command tells CCured to replace all references to `crypt` in a program with `crypt_wrapper`, which has the same interface. Note that this replacement occurs also in the places where the address of the function is taken. This new layer of indirection provides a convenient encapsulation for the constraints and run-time actions needed when calling this external function.

This wrapper checks that the first argument to the function is a null-terminated string, and that the second argument is a sequence of at least two characters. Then we strip the metadata from the CCured pointers and call the library function using standard arguments. Finally, we trust the library function to return a null-terminated string (or a null pointer). So we invoke the helper function `__mkptr_string`; when `thinResult` is nonnull, this routine constructs a CCured pointer with a home area that starts at "`thinResult`" and ends at "`thinResult + strlen(thinResult) + 1`." If there is a bug in the library that causes it to return a pointer to a buffer that has no null character, the wrapper will read past the end of that buffer just as the original program would have. CCured offers no protection against bugs in external functions.

Table 1. Some of the helper functions provided by CCured for use in wrappers. Here, "`void * SAFE`" refers to a standard (one-word) pointer that is compatible with external functions, while all other uses of "`void *`" refer to an arbitrary CCured pointer

Group A. These functions are for use on *CCured pointers*:

- `void * SAFE __ptrof(void *ptr);`
  Type inference: no constraints.
  At run time: returns `ptr`'s underlying standard pointer.
  Asserts: `ptr` is either null or within bounds.
- `int __check_string(char *ptr);`
  Type inference: `ptr` must carry an upper bound (i.e. it must be `FSEQ` or `SEQ`, not `SAFE`).
  At run time: checks that `ptr` points to a valid string, and returns the length of the string.
  Asserts: `ptr` is nonnull, within bounds, and points to a buffer that has a terminating nul character.
- `void __ensure_length(void *ptr, unsigned int n);`
  Type inference: `ptr` must carry an upper bound.
  At run time: verifies that `ptr` is at least `n` bytes long.
  Asserts: `ptr` is nonnull, within bounds, and has at least `n` bytes between the pointer and the end of the home area.

Group B. These functions are for use on *standard pointers* (i.e. those returned by the library) and do not perform any checking:

- `void * __mkptr(void * SAFE p, void *phome);`
  Type inference: phome must have the same kind (`SAFE`, `FSEQ`, etc.) as the return type.
  At run time: returns a (possibly fat) pointer to `p` with the same metadata as `phome`.
- `void * __mkptr_size(void * SAFE p, int size);`
  Type inference: no constraints
  At run time: returns a (possibly fat) pointer to `p` with the end of the home area equal to `p + size`.
- `char * __mkptr_string(char * SAFE p);`
  Type inference: no constraints
  At run time: returns a (possibly fat) pointer to `p` with the end of the home area equal to `p + strlen(p) + 1`. Equivalent to `__mkptr_size(p, strlen(p) + 1)`.

3.1 Curing with Wrappers

We have implemented wrappers for about 120 commonly-used functions from the C Standard Library. CCured inserts these wrappers into the relevant header files so that calls to these functions are handled correctly with no further intervention required. For example, whenever a program imports the "`crypt.h`" header it will automatically import `crypt_wrapper` as well. Programmers who work with

```
char *crypt(char const *key, char const *salt);
  ...
#pragma ccuredwrapper("crypt_wrapper", for("crypt"))
__inline static
char *crypt_wrapper(char const *key, char const *salt)
{
  __check_string(key);
  __ensure_length(salt, 2);
  char* thinResult = crypt(__ptrof(key), __ptrof(salt));
  return __mkptr_string(thinResult);
}
```

Fig. 2. A wrapper for `crypt`, a function that computes a hash of its arguments. The wrapper verifies interface assumptions using the `__check_string` routine, removes CCured-specific metadata with `__ptrof`, calls the underlying function, and packages up the result using `__mkptr_string`. Note that `crypt_wrapper` has the same signature as `crypt`

their own libraries can insert the wrappers anywhere in the program — as long as CCured sees the `#pragma` it will substitute the wrapper globally.

Most of these wrappers have a very small footprint, so we declare them as "`inline`" to specify that the body of the wrapper should be inlined at the call site. One case where the wrapper cannot be inlined is with function pointers — instead of taking the address of the original function, we will take the address of the wrapper function.

When CCured processes wrapper code, it will add all of the usual safety checks and instrumentation, in addition to the checks that the programmer explicitly requested with the helper functions. From CCured's perspective, the only difference between a wrapper function and a normal function is that when it sees a call to a library function in a wrapper, it will allow the code to call the underlying library. In a regular function, a call to an external library will be replaced by a call to the appropriate wrapper.

4 Context Sensitivity

CCured's kind inference mechanism is flow and context insensitive. All call sites of a function will be treated as assignments of actual arguments to the same formal argument. This insensitivity means that the pointer kind associated with a formal argument is the most conservative required by any call site. For example, if there is a call to `setFifth` that requires a `SEQ` return value, then variable b in `setFifth` will be inferred to be `SEQ`, and every call site of the function will be required to pass a `SEQ` argument. This can cause undesirable `SEQ` pointers to spread throughout a program. This problem can become even worse when a function is called with incompatible types at different call sites. CCured is forced in this case to use a more expensive dynamically-typed pointer which we do not discuss here. In fact such problems would almost certainly occur for

helper functions such as `__ptrof` that are intended to be used on any pointer type whatsoever.

To prevent this problem, CCured offers programmer-controlled context sensitivity. If the programmer declares a function to be context sensitive, CCured will treat each invocation of the function (or place where the address of the function is taken) independently.

Our wrappers use this context sensitivity for both the helper functions and the wrapper specifications themselves. This means that helper functions can operate on different pointer kinds when used in different contexts. We may infer that the argument and return type for `__mkptr` are `SAFE` in one location and `SEQ` in another. (Later, after the inference of kinds is complete, we replace the helper functions with code that actually performs the requested operation.) Similarly, the wrappers themselves can change depending on the call site.

CCured handles context-sensitive functions by creating a fresh copy of the function body for each call site, and then using its normal inference. When this is finished, CCured can coalesce any duplicate bodies back together, so that if some copies of `setFifth` use `SEQ` pointers and another group uses `FSEQ`, we need only two copies of `setFifth` in the final code — one for each kind. But coalescing can only go so far in preventing code bloat, so we usually disable context sensitivity for ordinary functions. We find that context sensitivity is more useful for wrappers than ordinary code because library functions (and hence their wrappers) tend to be used in multiple, unrelated parts of a program, making cross-contamination of constraints likely. Moreover, wrappers tend to be so small that they are often declared as "`inline`" already. By instantiating them at each call site early in the analysis, we can get the benefits of context sensitivity without further increasing the size of the code.

5 Deep-Copying Wrappers

The wrappers described so far work only when the library function accesses a single layer of pointers. Helpers such as `__ptrof` and `__mkptr` can manipulate metadata on the top layer, but are not enough when the library needs to access pointers to pointers. Because CCured changes the representation of pointers, data structures that include pointers are almost always incompatible with library code.

For most such data structures, we need a stronger wrapper mechanism. We therefore introduce *deep-copying wrappers*, which create a copy of the complex data structure in order to remove or add metadata. Deep-copying wrappers are regular wrapper that use some additional specification of how a data structure should be copied. When passing data to a library, we create a copy of the structure with no metadata; when retrieving data from a library we create a copy that has metadata to match its structure. (CCured's garbage collector insures that the newly-allocated copies are not leaked.) In exchange for breaking aliasing and adding a small performance cost, we have a means to exchange data between the two systems.

As an example, consider the following structure that is used for hostname lookups. For brevity, we show only three representative fields of the structure.

```
struct hostent {
  char  *      h_name;       /* official name of host */
  char  **     h_aliases;    /* alias list */
  short        h_addrtype;   /* host address type */
};
```

The library function `gethostbyname`, and related functions that perform domain name queries, return a pointer to a structure of this type. *h_name* points to a string containing the domain name and *h_aliases* points to a null-terminated array of string pointers representing other domain names for the computer.

CCured recognizes three pointer types for this structure, one for each use of "`*`". Each of these three pointer nodes may be given a different kind by the CCured inference depending on how they are used in the program. If all three pointers are inferred to be `SAFE`, then an ordinary wrapper will work fine. However, this outcome unlikely unless the *h_name* and *h_aliases* fields are never used, since reading any more than the first element requires array bounds information. If an ordinary wrapper were used when any of the pointers had a kind other than `SAFE`, a compilation error would occur to prevent the changed representation of `struct hostent` from causing bugs in the program.

We will consider the common case where each of these pointers becomes `SEQ`, reflecting the need for bounds to check the array accesses. This will change the shape of `struct hostent` in two ways, as shown in Figure 3:

1. The size of the struct itself will increase by four words, and the offsets of the fields will change, when the two top-level pointers are changed to wide pointers.
2. The inner pointers of *h_aliases* will also become wide pointers. This means that elements in the *h_aliases* array are now three words wide, so the array cannot be safely accessed by code that expects the array elements to be one word wide.

Either of these changes will pose a problem to our simple wrappers. Our solution is to define two versions of `struct hostent`: one that uses standard pointers, and one that uses wide. At runtime, we copy the data between the two representations.

5.1 Defining a Compatible Representation

Deep-copying wrappers use the `COMPAT` annotation to declare a type that should have no wide pointers. Figure 4 shows a wrapper that uses `COMPAT`.

When CCured sees this annotation, it creates two versions of the struct, which can be seen in Figure 5. The first, which keeps the name of the original struct (`hostent` in this case) has the default behavior for a type in CCured, and may be transformed to use wide pointers. The second definition will be given a new name

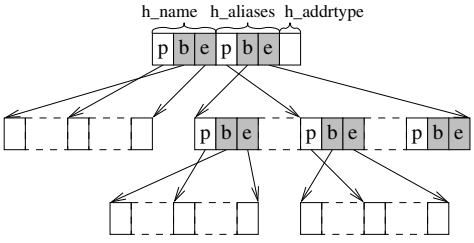

Fig. 3. Representation of `struct hostent` after the CCured transformation. Array-bounds metadata (gray) is interspersed with data (white)

```
#pragma ccuredwrapper("gethostbyname_wrapper", for("gethostbyname"));
__inline static
struct hostent* gethostbyname_wrapper(const char * name) {
  __check_string(name);
  struct hostent COMPAT * hcompat = gethostbyname(__ptrof(name));

  __DECL_NEW_FROM_COMPAT(hres, hostent, hcompat);
  return hres;
}
```

Fig. 4. A wrapper for `gethostbyname`, which performs a DNS lookup for the specified domain name. This wrapper validates its input, calls the underlying library function, and receives as a result a structure that uses standard pointers. `__DECL_NEW_FROM_COMPAT` allocates a new `struct hostent` called *hres* and populates it with the data in *hcompat*. (If *hcompat* is null, then *hres* will be too.) Section 5.2 describes how CCured generates the metadata needed by `__DECL_NEW_FROM_COMPAT`

(`hostent_COMPAT` in this case) and used wherever the programmer has specified the `COMPAT` declaration. The COMPAT version uses the same representation as C and never includes wide pointers, so this is the format that the library uses.

5.2 Generating Deep Wrappers

Now we need a mechanism to copy data between the wide and COMPAT structures. CCured generates a "deep copy" function to copy between the two representations when the wrapper author specifies, for each field of each struct, how such copying should be done. Depending on how the program interacts with the library, we may need to copy data from wide structs to standard, vice versa, or both.

For example, consider the standard-to-wide direction used for `gethostbyname`. Figure 6 shows the annotation that we give to CCured for this case.

- The `h_addrtype` field does not involve pointers, so CCured knows to copy this value directly without any annotation needed from the programmer.

```
struct hostent {
   char * SEQ        h_name;
   char * SEQ * SEQ  h_aliases;
   short             h_addrtype;
};
```

```
struct hostent_COMPAT {
   char *       h_name;
   char  **     h_aliases;
   short        h_addrtype;
};
```

Fig. 5. Possible definitions of `struct hostent` after CCured's transformations. `char * SEQ` is a wide pointer to an array of characters, and `char * SEQ * SEQ` is a wide pointer to an array of `char * SEQ`s

- We use the annotation on line 3 to say that the *h_name* field is a string. The generated deep copy function uses `__mkptr_string` to create a wide pointer to this buffer.
- We add the annotation on line 4 to say that *h_aliases* is a null-terminated array of strings. This is a fairly common data structure in C, and CCured includes code that will copy this specific case. For more complicated arrays, we would put C code in this function to do any copying or checking needed.

Similarly, CCured allows programmers to specify conversions for each field in a wide-to-standard deep copy. However, the wide-to-standard direction tends to be simpler, because we do not need the programmer to tell us how to create metadata. It is much easier to delete metadata than it is to invent it.

6 Experiences

The `libc` wrappers that are packaged with the CCured distribution are important for CCured's ease of use. Programmers who want to move new programs to CCured do not have to worry about modeling these library routines.

We can compare a hand-written wrapper with one generated automatically using our heuristics, but there is no truly sound way to verify the correctness of a wrapper. The programmer who writes a wrapper or checks the output of the generation tool may make a mistake; the documentation of the library may be incorrect; or a particular implementation of a library routine may be unsound. However, we have found that our system of wrappers works well in practice. The wrapper functions are simple and easy to read, and they have succeeded in

```
1 __inline static
2 __DEEPCOPY_FROM_COMPAT_PROTO(hostent) {
3     __DEEPCOPY_FROM_COMPAT_STRING_FIELD(h_name);
4     __DEEPCOPY_FROM_COMPAT_STRINGARRAY_FIELD(h_aliases);
5 }
```

Fig. 6. A specification that tells CCured how to do a deep copy of `struct hostent`. This specification must accompany the wrapper in Figure 4

finding several real-world bugs that involved improper invocations of a library routine.

In this section, we give examples of wrappers that are provided with CCured. Most of the wrappers we use are as simple as the examples we have shown so far, but we also give in this section some of our more complex wrappers, such as those for `open` and `qsort`.

```
1 #pragma ccuredwrapper("strchr_wrapper", for("strchr"))
2 __inline static
3 char* strchr_wrapper(char* str, int chr)
4 {
5    __check_string (str);
6    char* result = strchr(__ptrof(str), chr);
7    return __mkptr(result, str);
8 }
```

Fig. 7. A wrapper for `strchr`, a function that returns the first occurrence of character *chr* in string *str*

Figure 7 shows a wrapper for the `strchr` function. As before, we use a `pragma` command to specify that calls to `strchr` in the program should be replaced with calls to this wrapper. The `__check_string` helper function on line 5 checks that *str* is a null-terminated C string using a linear scan of the buffer. `__ptrof` strips the metadata from this pointer so that it can be passed to the underlying function. The result of `strchr` is a pointer to a character in the *str* buffer, so we use the `__mkptr` helper to create a new wide pointer by combining the return value of the library function with the metadata from *str*. Note that this helper function propagates constraints from the return value to the *str* argument, so CCured will ensure that the argument *str* has any metadata needed to construct the return value.

Figure 8 wraps the `open` function. `open` returns a handle for the file whose path is specified by the first argument. The second argument is a bit vector of options. If the `O_CREAT` option of oflag is set, then there is a required third argument: an integer specifying the permissions with which the new file should be created. Our wrapper for `open` matches the variable-argument interface of the library function, with the annotation on line 2 telling CCured that any additional arguments will be `int`s.

Our wrapper for `open` must do three things: check that the filename is a valid string, using `__check_string`; strip its metadata with `__ptrof` as usual; and check that the third parameter is present if it is required. We do this last check using the standard C macros for implementing variable-argument functions, as shown in lines 11–13 of the figure. However, when this wrapper is processed by CCured, the vararg macros will be replaced by typesafe code. The new version of `va_arg` checks that each argument was actually present at the call site before attempting to access it. This runtime check uncovered a bug in the OpenSSH daemon where the third parameter was missing from a certain call to `open`. C normally provides

```
1 #pragma ccuredwrapper("open_wrapper", for("open"));
2 #pragma ccuredvararg("open_wrapper", sizeof(int))
3 __inline static
4 int open_wrapper (char *file, int oflag, ...) {
5    __check_string (file);
6    if(oflag & O_CREAT){
7       //The O_CREAT flag is set, so the mode is required.
8       int mode;
9       va_list argptr;
10
11      va_start( argptr, oflag );
12      mode = va_arg( argptr, int );
13      va_end( argptr );
14
15      return open(__ptrof(file), oflag, mode);
16   } else {
17      return open(__ptrof(file), oflag);
18   }
19 }
```

Fig. 8. A wrapper for `open`

no mechanism for determining how many parameters are passed to a function, so the implementation of `open` simply reads the next word on the stack, and in this case was creating the file with garbage permissions.

For variable-argument functions that use a printf-like interface, we provide a special mechanism for checking the correctness of the argument list against the format string. This checking is done statically when possible, or at runtime if the format string is not a constant. The "`%n`" argument specifier is prohibited for security reasons [2], but all other features of printf are supported.

In Figure 9, we give a wrapper for the standard library's quicksort routine `qsort`. This function takes as input an array of values, and a pointer to a comparison function that will compare any two of the elements in the array. The library will make many calls to the comparison function for each array that it sorts.

This callback function makes wrapping `qsort` problematic. Since the comparison is user-defined, it will be processed by CCured and may need to use wide pointers, which it cannot get from the library. To fix this, we use a different sort of wrapper. The function "`__qsort_compare_wrapper`" takes standard arguments from a library and constructs fat pointers to pass to a CCured-processed function — the reverse of the usual process. In order to do this, we store a copy of the array pointer in global memory. Now we can use `__mkptr` inside the comparison wrapper to generate wide pointers.

The use of global variables to simulate a closure is a significant limitation of this wrapper, as it prevents the wrapper from having a polymorphic type. The `qsort_wrapper` and `qsort_compare_wrapper` functions can be made context sensitive, but there is no analogous mechanism for context sensitive global

```
1  static void *__qsort_base;
2  static int (*__qsort_compare)(void*, void*);
3
4  static
5  int __qsort_compare_wrapper(void * SAFE left, void * SAFE right)
6  {
7     // map the 'left' and 'right' lean pointers to the
8     // fat pointers we need
9     void* fatleft = __mkptr (left, __qsort_base);
10    void* fatright = __mkptr (right, __qsort_base);
11
12    // and call the user-supplied sorting function, which
13    // expects fat pointers
14    return __qsort_compare(fatleft, fatright);
15 }
16
17 #pragma ccuredwrapper("qsort_wrapper", for("qsort"));
18 __inline static
19 void qsort_wrapper(void* base,
20                    size_t nmemb,
21                    size_t size,
22                    int (*compare)(void *left, void *right))
23 {
24    __cleartags (base, nmemb * size);
25
26    // save the pertinent values
27    __qsort_base = base;
28    __qsort_compare = compare;
29
30    qsort(__ptrof (base), nmemb, size, __qsort_compare_wrapper);
31
32    __qsort_base = 0;
33 }
```

Fig. 9. A wrapper for `qsort`. While `qsort` is executing, two global variables store the metadata of the array being accessed and a reference to the user-defined comparison function, respectively. We pass "`__qsort_compare_wrapper`" to the library's `qsort` as the comparison function; when this intermediate function is called we use the global variables to reconstruct fat pointers for the relevant arguments and call the user's comparison function

variables. After all, code can be replicated without changing its behavior, but not so a variable. This wrapper will work so long as `qsort` is only used on arrays of one type and one kind, but any attempt to use it on multiple pointer types will result in a "bad cast" since CCured will notice that you assign arrays of different types to *__qsort_base*.

One solution would be to add a mechanism to CCured that would support true closures. However, we would prefer not to introduce that much complexity

into the language solely for use by callback functions. Our current distribution offers programmers a choice of two wrappers for `qsort`: this one, which may be used whenever the program only sorts lists of a single type; and a version without the wrapper for the comparison function, which can be used if the programmer carefully structures his comparison function so that it takes `SAFE` pointers as arguments.

7 Related Work

Several other systems have used wrapper functions to check that library functions are used properly. For example, Vo et al. [3] use wrapper functions written in a custom specification language to test for safe use of libraries, particularly with respect to error return codes. Ignoring errors returned by library functions is a common source of software problems. Their wrappers can automatically retry library functions that fail due to transient problems such as insufficient resources.

Libsafe [4] is a set of wrapper functions for the standard C library that attempts to prevent stack smashing attacks. When a stack buffer is passed to a library function, the libsafe wrapper can use the frame pointer to detect whether the library is at risk of overwriting a return pointer. This does not protect heap locations, however, and still allows other locations in the same stack frame to be overwritten by a buffer overflow. Either of these could allow an attacker to gain control of a system.

Fetzer and Xiao [5] present HEALERS, a system of automatically generated library wrappers that check error return codes and certain preconditions. The tool can infer the preconditions for a function by testing it on a range of inputs. However, the set of checks that HEALERS can perform is limited by the unsound nature of C. In an earlier paper [6], these authors describe some ways in which bounds data can be found for heap objects when checking library preconditions. Together with the libsafe strategy for checking stacks, this system can prevent many buffer overflow attacks on the stack and in the heap. But these heap checks are more costly than ours, and the system provides no way to perform typechecking of variable argument functions. Nonetheless, HEALERS could overcome these limitations if it were used in the context of a runtime system such as CCured.

Suenaga et al. [7] present an interface definition language for a situation similar to ours. Fail-Safe C is a typesafe C compiler that increases the size of both pointers an integers to two words. Rather than writing wrappers as C functions, Fail-Safe C programmers annotate the declarations of library routines with the necessary constraints, and the compiler generates the wrapper. This has the nice property that wrapper specifications in Fail-Safe C can be more concise than in CCured, and they avoid some "boilerplate" aspects of CCured's specifications such as the `#pragma`. However, CCured's wrappers give wrapper authors the full power and flexibility of the programming language. This flexibility allows us to wrap functions like `open` and `qsort`, which Fail-Safe C cannot, even though we

had not considered variable-argument functions or callback functions while we were initially designing our system.

Our deep-copying wrappers are unsatisfactory in some situations because the act of copying the data structures may break aliasing that is expected between the program and the library. An alternative to this copying is to store the metadata separately from the regular data, so that the regular data is in a format that is already compatible with the library. Recently, we have devised a solution that does exactly that [1]. Although it is possible that this split representation may cause slight performance degradation, it has performed well in practice. Even with a split representation, however, it is still necessary to have a wrapper that checks pre- and post-conditions.

A second alternative to the simple deep copy we propose is a call-by-copy-restore implementation, such as in [8]. Our current implementation provides call-by-copy semantics, which may break aliasing. With call-by-copy-restore semantics, we would copy the data to a compatible representation, call the library, and then copy modified data back to the original locations, where appropriate. This is more expensive than our naïve copying, but so long as the library does not retain a pointer to the data that is passed to it, these function calls would behave the same way as the original library calls do.

8 Conclusions

CCured provides an easy-to-use mechanism for integration with precompiled libraries. This mechanism is needed because CCured only operates on source code, and most programs rely on libraries whose source code is not readily available. Our solution supports both compile-time constraints and run-time instrumentation, and handles complex structures.

This system has helped us use CCured with a large number of real-world programs. Currently, we rely on wrappers to provide bug-checking and run-time support for over 120 functions. We use deep-copying wrappers for over a dozen functions, including `glob`, `getpwnam`, `gethostbyname`, and related functions. We believe that using C syntax for wrappers makes them easy to write and understand, and gives them sufficient power to check a wide range of constraints.

We have found several memory bugs in existing programs using these wrappers. Our system takes advantage of CCured's memory safety to perform more precise checks of certain preconditions than would normally be possible in a library. Among other things, we can check buffer lengths, null-terminated strings, and the arguments passed to variable-argument functions, including `printf`-like functions.

With this system, it is quick and easy to write wrappers and hook them into the system. Deep wrappers require little debugging, while other wrappers often require none at all. In practice, we have found some bugs in wrappers after they are first used – both bugs that allow memory errors to slip through and those that cause false positives. However these are not common, and the cost of

developing stable wrappers can be amortized over the many programs that use a library.

Acknowledgments

We would like to thank the other members of the CCured team — Jeremy Condit, Scott McPeak, and Westley Weimer — for their help with the development and testing of the system.

References

1. Condit, J., Harren, M., McPeak, S., Necula, G.C., Weimer, W.: CCured in the real world. In: Proceedings of the ACM SIGPLAN 2003 conference on Programming Language Design and Implementation, ACM Press (2003) 232–244
2. Newsham, T.: Format string attacks (2000) http://www.lava.net/~newsham/format-string-attacks.pdf.
3. Vo, K.P., Wang, Y.M., Chung, P.E., Huang, Y.: Xept: a software instrumentation method for exception handling. (1997) 60–69
4. Baratloo, A., Singh, N., Tsai, T.: Transparent run-time defense against stack smashing attacks. In: Proceedings of the USENIX Annual Technical Conference. (2000)
5. Fetzer, C., Xiao, Z.: An automated approach to increasing the robustness of C libraries. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, IEEE Computer Society (2002) 155–166
6. Fetzer, C., Xiao, Z.: Detecting heap smashing attacks through fault containment wrappers. In: Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems. (2001)
7. Suenaga, K., Oiwa, Y., Sumii, E., Yonezawa, A.: The Interface Definition Language for Fail-Safe C. In: Proceedings of the 2003 International Symposium on Software Security. Lecture Notes in Computer Science, Springer (2004)
8. Tilevich, E., Smaragdakis, Y.: NRMI: Natural and efficient middleware. In: Proceedings of the 23rd International Conference on Distributed Computing Systems, IEEE Computer Society (2003) 252