Debugging via Run-Time Type Checking

Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps

Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison, WI 53706 USA
{alexey, suan, horwitz, reps}@cs.wisc.edu
Fax: (608) 262-9777

Abstract. This paper describes the design and implementation of a tool for C programs that provides run-time checks based on type information. The tool instruments a program to monitor the type stored in each memory location. Whenever a value is written into a location, the location's run-time type tag is updated to match the type of the value. Also, the location's static type is compared with the value's type; if there is a mismatch, a warning message is issued. Whenever the value in a location is used, its run-time type tag is checked, and if the type is inappropriate in the context in which the value is being used, an error message is issued. The tool has been used to pinpoint the cause of bugs in several Solaris utilities and Olden benchmarks, usually providing information that is succinct and precise.

1 Introduction

Java programmers have the security of knowing that errors like out-of-bounds array indexes or attempts to dereference a null pointer will be detected and reported at runtime. Java also provides security via its strong type system. For example:

- There are no union types in Java, so it is not possible for a program to write into a field of one type and then access that value via a field of a different type.
- Only very restricted kinds of casting are allowed; for example, it is not possible to treat a pointer as if it were an integer or vice versa.
- When an object is down-cast to a subtype, a run-time check is performed to ensure that the actual type of the object is consistent with the cast.

C and C++ programmers are not so lucky. These languages are more liberal than Java in what they allow programmers to express; the static type system is weaker; and the run-time system provides little in the way of protection from errors caused by misuse of casts, bad pointer dereferences, or array out-of-bounds errors. Programmers can use Purify[1], Safe-C[2], and shadow processing[3] to help detect bad memory accesses, but those tools provide no help with the many additional kinds of errors that can be introduced into C and C++ programs due to their weak type systems.

This paper describes the design and implementation of a tool for C programs that performs run-time checks based on type information. The tool instruments a program to monitor the type stored in each memory location (which may differ from the static type of that location due to the use of unions, pointers, and casting). Whenever a value is written into a location, the location's run-time type tag is updated to match the type of the value. Also, the location's static type is compared with the value's type; if there is a mismatch, a *warning* message is issued. Whenever the value in a location is used, its run-time type tag is checked, and if the type is inappropriate in the context in which the value is being used, an *error* message is issued.

The tool is able to find all of the run-time storage violations found by Purify (e.g., a use of an uninitialized variable or an out-of-bounds array access). In these cases, the tool's error messages are roughly equivalent to those reported by Purify on a given run of a faulty program. The warning messages, however, provide more information about what occurred prior to the error, which can be of great help when trying to identify the statements that actually caused the error. In addition, the tool has the potential to find errors that Purify cannot detect (e.g., a write into one member of a union followed by a read from a member of a different type).

In preliminary tests, the tool has been used to find bugs in several Solaris utilities and Olden benchmarks. The information provided by the tool is usually succinct and precise in showing the location of the error.

The remainder of the paper is organized as follows: Section 2 provides several examples that illustrate how the tool works and what kinds of errors it can detect; Section 3 describes a preliminary implementation of the tool; Section 4 discusses the results of some experiments; and Section 5 concerns related work.

2 Motivating Examples

In this section, we provide three motivating examples to illustrate the potential benefits of providing run-time type checking. In each case, we describe the kind of error that might be made, how our tool would detect the error at run-time, and the interesting issues raised by the example.

2.1 Bad Union Access

A very simple example of a logical error that manifests itself as a bad run-time type is writing into one field of a union and then reading from another field with a different type. This is illustrated by the following code fragment:

```
union U { int u1; int *u2; } u;
int *p;
u.u1 = 10; /* write into u.u1 */
p = u.u2;  /* read from u.u2 – warning! */
*p = 0;    /* bad pointer deref – error! */
```

In this example, an integer value is written into variable `u` (on line 3), and is subsequently read as a pointer (on line 4). The value that is read from `u` is stored in variable `p`, which is then dereferenced (on line 5). The symptom of the error is the attempt to use the value 10 as an address on line 5; however, the actual point of the error can be said to be on line 4, when a value of one type is read as if it were another type (i.e., the run-time type of `u.u2` is not the same as its static type).

A tool like Purify would report an error when line 5 was executed; however, it would not be able to point to line 4 as the source of the error.

Recall that our tool instruments the program to track the run-time types of memory locations. In the example, the single location that corresponds to both `u.u1` and `u.u2` would have an associated run-time type. That type would be set to `int` after the assignment `u.u1 = 10` on line 3. On line 4, the location is read, and its value is assigned to a pointer; this is a type mismatch, and therefore our tool would produce a warning message when line 4 is executed (as well as an error message reporting the run-time type violation at line 5).

2.2 Heterogeneous Arrays

C programmers sometimes try to avoid the overhead of the `malloc` and `free` functions by writing their own dynamic memory-management functions. For example, a programmer might allocate a large chunk of memory using a single call to `malloc` via an assignment like the following:

```
char *myMemory = (char *)malloc(BLOCKSIZE);
```

(where `BLOCKSIZE` is some large integer value). Subsequently, when new memory is needed, a call to a user-defined function, e.g., `myMalloc`, is made, rather than a call to `malloc`. The `myMalloc` function returns a pointer to an appropriate part of the `myMemory` "chunk". Similarly, calls to `free` are replaced by calls to `myFree`, which updates appropriate data structures to keep track of which parts of `myMemory` are currently in use.

The `myMemory` "chunk" could be used as a heterogeneous array; i.e., different parts of the array could contain values of different types. For example, the programmer's code might include the following declarations and calls:

```
1. struct node { int data; struct node *next; } *n, *tmp;
2. int *p = (int *) myMalloc(100 * sizeof(int));
3. n = (struct node *) myMalloc(sizeof(struct node));
```

The call on line 2 allocates an array of 100 integers, and the call on line 3 allocates one node for a linked list.

Now suppose that there is a bug in the programmer's memory-allocation code that causes it to return overlapping chunks of memory. In particular, assume that the value assigned to variable `n` on line 3 is the same as the address of `p[98]`. In addition, assume that pointers and integers both take 4 bytes, and that there is no padding between the two fields of `struct node`. In this case, after the call to myMalloc on line 3, the address of n->data is the same as the address of p[98], and the address of n->next is the same as the address of p[99]. Now consider what happens when the following statements are executed:

```
4. n->next = (struct node *) myMalloc(sizeof(struct node));
5. p[99] = 0;
6. tmp = n->next;
```

Since p[99] and n->next refer to the same location, the assignment on line 5 overwrites the value assigned to n->next on line 4 with the value 0, essentially replacing the link to the next node in the list with a (list-terminating) NULL. Therefore, future accesses to the list will find only one node. If the assignments on lines 4 and 5 were in different parts of the code (e.g., in unrelated functions) the source of this error might be very difficult to track down (and a tool like Purify would not be able to help, since there are no bad pointer dereferences or array-access errors. Of course, if the assignment on line 5 set p[99] to some value other than zero, then future accesses to the list would probably cause a bad pointer dereference, which would be detected by a tool like Purify. However, as in the "bad union access" example above, Purify would not be able to locate the source of the error.)

Our tool would tag the elements of myMemory with their run-time types. For example, after the assignment on line 4, the location that corresponds to n->next would be tagged with type pointer. The assignment on line 5 would change that tag to int. Finally, the use of the value in n->next on line 6 would cause a warning message to be reported, because the location is tagged with run-time type int while its value is being assigned to a pointer (tmp).

2.3 Using Structures to Simulate Inheritance

C is not an object-oriented language, and therefore has no classes. However, programmers often try to simulate some of the features of classes using structures[4]. For example, the following declarations might be used to simulate the declaration of a superclass Sup and a subclass Sub:

```
struct Sup { int a1; int a2; };
struct Sub { int b1; int b2; char b3; };
```

A function might be written to perform some operation on objects of the superclass:

```
void f ( struct Sup *s ) {
   s->a1 = ...
   s->a2 = ...
}
```

and the function might be called with actual arguments either of type struct Sup * or struct Sub *:

```
struct Sup sup;
struct Sub sub;
f(&sup);
f(&sub);
```

The ANSI C standard guarantees that the first field of every structure is stored at offset 0, and that if two structures have a common initial sequence – an initial sequence of one or more fields with compatible types – then corresponding fields in that initial sequence are stored at the same offsets. Thus, in this example, fields `a1` and `b1` are both guaranteed to be at offset 0, and fields `a2` and `b2` are both guaranteed to be at the same offset. Therefore, while the second call, `f(&sub)`, would cause a compile-time warning (which could be averted with an appropriate type cast), it would cause neither a compile-time error nor a run-time error, and the assignments in function `f` would correctly set the values of `sub.b1` and `sub.b2`.

However, the programmer might forget the convention that `struct Sub` is supposed to be a subtype of `struct Sup`, and might change the type of one of the common fields, might add a new field to `struct Sup` without adding the same field to `struct Sub`, or might add a new field to `struct Sub` before field `b2`. For example, suppose the declaration of `struct Sub` is changed to:

```
struct Sub { int b1; float f1; int b2; char b3; };
```

Now, when the second call to `f` is executed, the assignment `s->a2 = ...` would write into the `f1` field of sub rather than into its `b2` field. The fact that the `b2` field is not correctly set by the call to `f`, or the fact that the `f1` field is overwritten with a garbage value will probably either lead to a run-time error later in the execution, or will cause the program to produce incorrect output.

Once again, the use of run-time types can help. The assignment `s->a2 = ...` causes `sub.f1` to be tagged with type `int`. A later use of `sub.f1` in a context that requires a `float` would result in an error message due to the mismatch between the required type (`float`) and the current run-time type (`int`).

Note that in this example, a tool like Purify would not report any errors, because there are no bad pointer or array accesses: function `f` is not writing outside the bounds of its structure parameter, it just happens to be the wrong part of that structure from the programmer's point of view.

3 Implementation

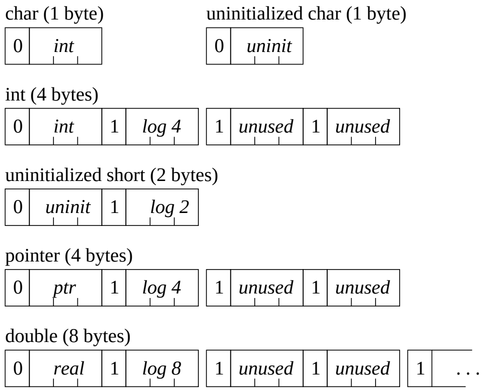
Our debugging tool has been implemented for all of ANSI C. It has two major components: a compiler front-end that instruments the program, and a run-time system that tracks the dynamic type associated with each memory location.

3.1 Tracking Type Information

The run-time types that are tracked are: *unallocated*, *uninitialized*, *integral*, *real*, and *pointer*. For aggregate objects (structures and arrays), each field/element has its own tag. For pragmatic reasons, typedefs are ignored, and different pointer types are not distinguished. While it might be useful to distinguish different user-defined types and different kinds of pointers, doing so would mean having no *a priori* upper bound on the number of types in a program, and therefore no upper bound on the number of bits required to represent a type, which would greatly complicate run-time type checking.

The run-time component of our tool is implemented by storing type information in a "mirror" of the memory used by the program. Each byte of memory maps to a four-bit nibble in the mirror. Of these four bits, one "continuation" bit encodes the extent of the object (0 denotes the start of a new object, 1 denotes a "continuation" nibble), and three "data bits" encode other information. In the first nibble of an object's tag, the "data bits" encode the object's current type; in the second nibble (if the object is larger than one byte in size) the "data bits" encode (log_2 of) the size of the object. This scheme allows for quick comparisons between two objects by merely comparing the first eight bits (two *nibbles*) of their tags. For objects larger than two bytes, the remaining "data bits" are currently unused (they may potentially be used to encode information for future enhancements or optimizations).

The tags for some common scalar types (with their sizes) are illustrated below:



The mirror is allocated in segments corresponding to 1MB of user memory, as the amount of memory in use by the program increases. Pointers to these "mirror pages" are stored in a table indexed by the most significant 12 bits of the user-space address, so accesses to an object's tag are fast. The interface to the run-time system consists primarily of procedures (implemented using macros whenever possible to cut down on the overhead of function calls) that set a tag (`setUninitTag` and `setScalarTag`), copy a tag (`copyTag`), and verify that a tag agrees with an expected type (`verifyTag`). There is also a procedure (`verifyPtr`) to verify that a pointer points to allocated memory before it is dereferenced, and a set of procedures to handle the passing of function parameters and return values (`processArgTag` and `processReturn`).

3.2 Source-Level Instrumentation

To instrument a program, the tool performs a source-to-source transformation on the C source files using Ckit[5], a C front end written in ML. Working at the source level gives the tool access to all the source-level type information it needs. Also, the flexibility of the comma operator in C makes it possible to preserve the ANSI C semantics of the original program while retaining portability: an instrumented C file can, in principle, be compiled on multiple platforms.[1]

Handling the C language was non-trivial because a number of C's features make correct instrumentation difficult. The following summary of the instrumentation actions that the tool performs highlights some of the issues:

main: Every occurrence of `main` is renamed to `prog_main`. Our run-time system defines its own `main` function, which performs some initialization before calling `prog_main`. This way, we can filter out command-line arguments for the run-time system, and initialize the tags for the `argv` arrays. Also, this way recursive calls to `main` do not cause any problems.

statements and expressions: Each program statement and its component expressions are instrumented via a set of syntax-directed transformations. Code for setting, copying, or verifying tags is added to expressions; the instrumented code makes extensive use of the comma operator (see Section 3.3 for an example).

locals: Local variables are initially tagged *uninitialized*. A local variable that is initialized is processed as if the initialization expression were assigned to that variable. Because the instrumentation code needs to be able to take the address of all variables, `register` variables are demoted to `auto` variables. Also, tags for bit fields are not initialized because C does not allow the address of a bit field to be taken; instead, bit fields remain tagged as *uninitialized*. This still triggers type-violation warnings and errors if a bit field is used in a type-unsafe manner.

globals: Tags for global variables are initialized in a special `init` function; one such function is created per source file. Our `main` function calls each of these `init` functions before calling `prog_main`. The list of `init` functions from the different source files is collected at link time.

externs: `extern` variables that are not defined in any of the instrumented source files are treated specially. To allow instrumented source files to be linked with uninstrumented object code (most commonly library modules), we assume that `extern` variables are "well-behaved", and so initialize their tags to contain their declared types. However, the tool is limited by what is visible to it. In particular, it cannot initialize the tags for incomplete array types (e.g., `extern int a[];`) because the size of the array is not visible.

calls: To handle function calls, the tags of the function's parameters must be communicated between the caller and the callee. At the callsite, code is added to store the tags for the actual parameters in an array, whose address is kept in a global pointer, `globalArgTags`. At the head of the function definition, code is

[1] Note: the Ckit front end does not currently support C-preprocessor directives, so at present we can only instrument *preprocessed* C code. This limits portability to some extent, but is not a fundamental limitation of our approach.

added to extract the tags of the parameters passed to the function. The same mechanism is used to pass the tag of the return value back to the callsite. To allow a mix of instrumented and uninstrumented functions to work properly, including where instrumented functions are invoked via callbacks from uninstrumented functions, the instrumented caller stores the address of the callee in a global pointer, `globalCallTarget`, while the instrumented callee compares its own address with `globalCallTarget`. If the addresses match, it means that the caller is instrumented, so the tags for function arguments and return value are processed as described above. If the addresses do not match, however, it means that the caller is uninstrumented, so tags for function parameters cannot be extracted from `globalArgTags`.

return: At a return statement and at the end of a function *f*, the mirror for the entire stack frame of f must be cleared to *unallocated*. This is done by `processReturn`, a procedure in our run-time system. The start of the stack frame for f is assumed to be the greatest[2] of the addresses of f's formal parameters (if any) and the first local variable declared in f (a variable specially added by our instrumentation process). Since the call to `processReturn` itself has advanced the stack-frame pointer beyond the end of f's stack frame, a lower bound on the end of f's stack frame is obtained by taking the address of a local variable declared within `processReturn` itself.

3.3 Instrumentation Example

A C program is instrumented by transforming the statements contained in the bodies of the program's functions. Expressions in each statement are replaced by the result of a call to the instrumentation function *instr-expr*, which takes as arguments (1) *exp*, the expression to be instrumented, (2) a boolean flag *enforce*, and (3) an optional pointer argument *tagptr*.

The *enforce* flag specifies whether the expression's run-time type must match its static type, i.e., if the expression is used in a context that is sensitive to its type. For example, in the expression $e_1 + e_2$, the data in both e_1 and e_2 need to reflect their respective static types, so *enforce* must be *true* when instrumenting both e_1 and e_2. On the other hand, for the expression `&`e, the type of the data in e is not relevant to the type-safety of `&`e, so e would be instrumented with *enforce=false*.

The presence of the optional pointer argument *tagptr* indicates, when instrumenting an expression e, that an enclosing expression needs access to e's type. In such a case, *tagptr* is used to convey this information.

The output of applying *instr-expr* to three common expressions ($e_1 = e_2$, *id*, and $*e$), is shown in Table 1.[3]

[2] Assuming that the stack grows downwards in memory (from high to low addresses). For stacks that grow upwards, we use the lowest of the addresses of f's formal parameters and its first local variable.

[3] We omit some details that would only complicate the example. For instance, we actually perform slightly different actions for instrumenting lvalues and rvalues. Also,

Table 1. Examples of instrumentation rules

exp	***instr-expr*(*exp*,*enforce*)**	***instr-expr*(*exp*,*enforce*,*tagptr*)**
id	*(verifyTag(&*id*, *typeof*(*id*)),[1] &*id*)	*(*tagptr* = &id, verifyTag(&*id*, *typeof*(*id*)),[1] &*id*)
$*e$	*($\texttt{tmp}_{ptr}$ = *instr-expr*(*e*, *true*), verifyTag($\texttt{tmp}_{ptr}$, *typeof*($*e$)),[2] $\texttt{tmp}_{ptr}$)	*(*tagptr* = *instr-expr*(*e*, *true*), verifyTag(*tagptr*, *typeof*($*e$)),[2] *tagptr*)
$e_1 = e_2$	($\texttt{tmp}_{assign}$ = *instr-expr*(e_1, *false*, $\texttt{tmp}_{ptr1}$) = *instr-expr*(e_2, *enforce*, $\texttt{tmp}_{ptr2}$), copyTag($\texttt{tmp}_{ptr1}$, $\texttt{tmp}_{ptr2}$, *typeof*(e_1)), $\texttt{tmp}_{assign}$)	($\texttt{tmp}_{assign}$ = *instr-expr*(e_1, *false*, *tagptr*) = *instr-expr*(e_2, *enforce*, $\texttt{tmp}_{ptr2}$), copyTag(*tagptr*, $\texttt{tmp}_{ptr2}$, *typeof*(e_1)), $\texttt{tmp}_{assign}$)

[1] omit if *enforce* = *false*
[2] call verifyPtr instead if *enforce* = *false*

The tmp variables that appear in the rules in Table 1 are temporaries (of appropriate type) introduced by the instrumentation code. The instrumented expression is shown in the second and third columns: column two shows how instrumentation is carried out when the optional third argument is absent; column three shows the instrumentation strategy when the third argument, *tagptr*, is present. At run-time, the *tagptr* variable will be set to point to an object whose mirror is tagged with the expression's dynamic type, to be used in the instrumentation code of an enclosing expression (see the rules for $e_1 = e_2$). When *enforce* is *true*, the verifyTag procedure is used to verify that the tag associated with a given object agrees with its declared type.

For the *id* case, the only check done (when *enforce* = *true*) is to verify that *id*'s dynamic type agrees with its declared type.

For the dereference case, the subexpression e is first instrumented by passing *enforce* = *true* to *instr-expr* (since e will be dereferenced, i.e., used as a pointer). After that, if *enforce* = *true*, we verify that the dynamic type of $*e$ agrees with its declared type. If *enforce* = *false*, we do not require that $*e$'s dynamic type match its declared type; however, we still want to make sure that $*e$ is not *unallocated* (i.e., that e points to valid memory). This is performed by the verifyPtr procedure, which allows the tool to output an error message before an invalid pointer dereference occurs.

In the assignment case, expression e_1 is instrumented with *enforce* = *false*, since we do not care about the type of the data that is about to be overwritten; e_2 is instrumented with *enforce* = *true* only if the assignment expression is being instrumented with *enforce* = *true*. The copyTag procedure copies the tag of the right-hand-side expression to the mirror of the left-hand-side expression, and

if an expression's lvalue or rvalue is not used in a larger context, we avoid preserving that value in the instrumented expression.

also issues a warning message if the type of the right-hand-side expression is not compatible with the static type of the left-hand-side expression.

For the *id* and $*e$ cases, the instrumented code has the form `*(...,`*ptr*`)`; this is so that the instrumented expression is a valid lvalue. An assignment expression is not an lvalue, and so does not need to be instrumented in this way. However, if an assignment is used as an rvalue in a larger context, we must still make sure that the instrumented expression preserves the correct rvalue, which is the purpose of $\texttt{tmp}_{assign}$.

Now consider instrumenting the statement `x = *p;`. Since this assignment does not occur in a context that uses its value, its type need not be verified, and the assignment is instrumented with *enforce* = *false*. Also, since there is no enclosing expression that needs access to the assignment's type, the optional *tagptr* argument is not needed. Assuming `x` is of type `int` and `p` is of type `int *`, the result of *instr-expr*(`x = *p`, *false*) is shown in Figure 1. Notice that the `tmp` variable at the outermost level (for the assignment expression) has been omitted because in this case, the assignment's rvalue does not need to be preserved.

```
*(tmp1 = &x, &x) =
   *(tmp2 = *(verifyTag(&p, pointer_type),
               &p),
     verifyPtr(tmp2, sizeof(int)),
     tmp2),
copyTag(tmp1, tmp2, int_type)
```

Fig. 1. Output of *instr-expr*(`x = *p`, *false*)

3.4 Other Features

In order to perform proper type checking, the tool needs to handle memory-management functions specially. We replace each call to `malloc` (and its relatives) with our own version that, upon successfully allocating a block of memory, initializes the mirror for that memory block with the *uninitialized* tag. Similarly, our `free` function resets the mirror to be of *unallocated* type. Our versions of these functions do their own bookkeeping so we know how many bytes are being freed by a call on `free` at run-time. Our version of `malloc` also adds padding between allocated blocks to decrease the likelihood of a stray pointer jumping from one block to another (this is the approach used by Purify). For the stack allocation function `alloca`, we instrument the callsites to additionally invoke procedures to initialize the newly allocated stack space.

Another routine that is handled specially is the variable argument routine `va_arg` (usually implemented as a macro). Our portable solution assumes that the argument obtained is properly typed, so we instrument each invocation of `va_arg` to return a tag of the expression's static type.

As mentioned in Section 3.2, the approach we have taken allows us to link instrumented modules with uninstrumented ones, with the only requirement being that the program's `main` function must be renamed to `prog_main`. This flexibility is useful if, for example, a programmer only wants to debug one small component of a large program: they can instrument just the files of interest, and link them in with the other uninstrumented object modules. A caveat when doing this, however, is that it may lead to the reporting of spurious warning and error messages because the uninstrumented parts of the code do not maintain type information. For example, if a reference to a valid object in the uninstrumented portion of the program is passed to an instrumented function, the tool will think that the object is unallocated, and may output spurious errors and warnings.

This problem extends, in general, to library modules. For example, the flow of values in a function like `memcpy`, the initialization of values from input in a function like `fgets`, and the types of the data in a static buffer returned by a function like `ctime` would not be captured. To handle these, we have created a collection of instrumented versions of common library functions that affect type flow. These are wrappers of the original functions, hand-written to perform the necessary tag-update operations to capture their type behavior.

Finally, our tool lends itself naturally to interactive debugging. When a warning or error message is issued, a signal (`SIGUSR1`) is sent, and can be intercepted by an interactive debugger like GDB[6]. The user is then able to examine memory locations, including the mirror, and make use of GDB's features to better track down the cause of an error.

4 Experiments

To test the effectiveness of our debugging tool, we used Fuzz[7] to find Solaris utilities that crash on some inputs, and instrumented five such programs for testing (`nroff`, `plot`, `ul`, `units`, `col`). We also tracked down bugs that appear in two programs from the Olden benchmark suite (`health`, `voronoi`). A summary of what our tool revealed about these runs is given below.

nroff: An array of pointers is accessed with a negative index, and the retrieved word, when dereferenced, causes a segmentation fault. The instrumented program, before crashing, warns that the retrieved word that is about to be dereferenced actually contains an array of characters.

plot: A rogue pointer, after passing beyond the bounds of a local array, walks up the stack, writing bytes as it goes. It eventually attempts to write to invalid memory, at which point the program crashes. The instrumented program outputs a long list of warning messages signaling these writes to unallocated memory, accurately identifying the line of code where this occurs.

ul: The original program crashes during a call to `fgetwc`, while the instrumented program crashes during a call to `fprintf` as our instrumentation code is attempting to write a warning message. The cause of the crash in the original program was difficult to diagnose, but "accessing unallocated memory" error messages generated by our instrumented program led us to the cause of the crash: a pointer, after passing beyond the bounds of an array, walks through the bss section and eventually overwrites part of the global `_iob` array (which contains information about `stdin`, `stdout`, and `stderr`). This causes the subsequent call to `fgetwc` in the original code, and `fprintf` in the instrumented code, to crash.

units: In the original program, an errant pointer manages to corrupt the "save" area of the call stack, resulting in bizarre behavior that was difficult to track down without our tool. The instrumented program issues a type-violation error message after the character pointer `cp` is set to point to itself, and is subsequently used to write a character value onto itself. The next dereference of `cp` generates another error message, and then the program crashes.

col: The original program crashes on a dereference of a bad pointer, but our instrumented program does not crash; instead, it fails to terminate (at least, after two hours we stopped waiting for it to terminate). The first of many error messages generated by our instrumented program signals a dereference into unallocated memory, and points to the line in the program where the crash occurs (in the uninstrumented code). The point where the error message was generated is close to where the pointer first stepped out of bounds of the global array to which it pointed.

health: In the semantics of C, memory allocated by `malloc`, unlike `calloc`, is not required to be zero-initialized, although many programmers assume that it is. In this program, the pointer fields in two recursive data structures are not initialized after allocation via `malloc`. While traversing these structures, the original program counts on the pointer fields being `NULL` to indicate the absence of a substructure. The instrumented program warns of an access to uninitialized memory each time the program checks to see if one of these pointer fields is `NULL`. On our testing platform, all the memory allocated with `malloc` in this program happen to be zero-initialized and so the program does not crash. However, the erroneous assumption about `malloc` is a program flaw that may cause a crash on a different execution (or when the program is run on a different platform).

voronoi: Some bit-level manipulations are performed on a pointer to a struct, yielding a pointer to a "field" that does not belong to the struct, since some assumptions made by `voronoi` about the size of the struct do not hold on our test machine. A subsequent assignment of this pointer (as a function argument) generates a warning message stating that an unallocated object is being passed. Later, when the pointer (which happens to be `NULL`) is actually dereferenced, the instrumented program gives an "accessing unallocated memory" error message before crashing.

In most cases, crashes in the test programs were found to have been caused by a pointer (or array index) that had gone astray. In every case, our tool was able to detect the out-of-bounds memory accesses because the type of the pointed-to memory was different from the expected type. While these results are very encouraging, these kinds of errors would also be detected by Purify.

We can easily create examples (such as the ones given in Section 2) for which our tool is able to detect errors that are *not* detected by Purify; however, we have not yet found examples of those kinds of bugs in real programs. We suspect that such bugs are more likely to occur in larger, more complicated programs, but due to limitations of the current version of the Ckit front end, we have not been

Table 2. Performance on the benchmarks ("Lines of C code" reports the number of unpreprocessed lines of source code, with comments and blank lines removed.)

source	program	lines of C code	running time (secs) uninstru-mented	running time (secs) instru-mented	slow-down
Olden bench-marks	bh	1,049	8.97	910.02	101.4
	bisort	570	7.60	70.86	9.3
	em3d	414	1.79	31.35	17.5
	health	559	6.45	56.97	8.8
	mst	493	3.25	83.10	25.6
	perimeter	389	2.22	49.13	22.1
	power	679	6.41	241.83	37.7
	treeadd	291	3.90	62.17	15.9
	tsp	567	12.78	83.64	6.5
SPEC bench-marks	compress	1,491	19.87	695.83	35.0
	gcc	151,531	11.08	1288.64	116.3
	go	26,917	12.04	654.86	54.4
	li	6,272	5.47	320.99	58.7
	m88ksim	14,502	1.80	239.91	133.0
	vortex	52,624	12.37	1596.02	129.1
Solaris utilities	col	502	1.47	29.39	20.0
	nroff	11,018	0.82	53.01	64.6
	plot	326	1.02	5.90	5.8
	ul	468	0.33	1.87	5.6
	units	457	0.39	3.19	8.2

able to successfully compile many large programs. Furthermore, the code that we have used to date for testing our technique is in most cases robust code that has been in use for quite some time. As a result, the likelihood of finding errors is lower than if the tool were applied to code during the software-development cycle.

Not surprisingly, the extensive checking performed by our tool comes at a performance cost. This cost is due to the execution of our type-tracking procedures, as well as to the transformation of the original program's expressions into more complicated ones in order to allow type tracking while preserving the original expressions' values, types, and side-effects. To measure the execution-time overhead that is introduced by our tool, we instrumented the five Solaris utilities described above, as well as several programs from the SPEC and Olden benchmarks. The benchmarks were optimized with gcc's -O2 option,[4] and executed with legitimate (non-crashing) inputs on a 300 MHz Sun Ultra 10 workstation with 256 MB of RAM and 1.1 GB of virtual memory. The sizes of the

[4] Except for `gcc`, for which the optimized instrumented version behaved differently from the original version, for unknown reasons.

benchmarks, as well as the execution times (user+system time, in seconds) and slowdowns are reported in Table 2.

The slowdowns we observe on these benchmarks range from about 6 times to 130 times, with a median of 23.8. As a point of comparison, the slowdown factor for Purify tends to be in the range of 10 to 20. The exorbitant slowdown for `bh` is due mainly to the program making many assignments of structures, for which our tag-copying procedure `copyTag` performs an inefficient nibble-by-nibble copy. For the SPEC benchmarks, as well as for `nroff`, the performance degradation is largely due to the overhead of writing out spurious warning and error messages. These mainly result from the tool's inability to cleanly capture the type behavior of `calloc`-initialized memory and incomplete array types (in particular, the `ctype` array, which is used by `ctype.h` macros), and also from some of these programs performing masking operations which treat integers as arrays of characters – technically a type violation.

Future work includes using the results of static analysis to reduce the amount of instrumentation introduced by our tool (thus reducing its overhead). For example, if the value in a location is used multiple times, and there is no possibility that its type is modified between the uses, then only the first use needs to be checked.

5 Related Work

Run-time type-checking is not a new idea. It has been implemented in functional-style languages like LISP and Scheme, and object-oriented languages like Java, Smalltalk and Objective-C. However, designing dynamic type-checking for a language like C that includes unions, structures, arrays without bounds checking, casting and pointer arithmetic is a large undertaking. Additionally, there is a subtle difference in approach. Traditional dynamic type-checking attaches tags to data, while our approach separates the tag space from the data space. Both approaches attain good spatial locality of accesses (in the case of separated tags and data, the locality of tag accesses mirrors the locality of data accesses). However, if checking of some of the tags can be eliminated based on static analysis, the locality of accesses to data does not suffer as it does in the case of co-located tags and data.

Another significant advantage is the fact that the tags can be protected from erroneous accesses by the user code. User code is simply less likely to make erroneous accesses to the separated tag space, and if better guarantees are required, memory-protecting hardware can be used to shield the tag space during the execution of user code.

The concept of soft typing [8, 9] was introduced in an effort to combine the benefits of static and dynamic typing for functional-style languages. In this approach, static typing is employed to identify program statements that do not statically type check. These statements are subsequently instrumented to be dynamically type-checked. The earlier work [8] concentrates on presenting a framework for soft typing on a restricted class of programming languages; the latter work [9] extends this work to handle realistic languages like Scheme. By addressing a traditionally dynamically typed language, the emphasis was put on the application of soft typing as a way of lowering the run-time overhead of type-checking. While this body of work is similar to ours in its approach to type-checking, the differences in the languages handled by the two approaches is significant.

Approaches to detection of errors in C programs by means of executing a program instrumented to perform run-time checks have been developed in the past. Safe-C[2] provides run-time detection of array access and pointer dereference errors, such as array out-of-bounds errors, stale-pointer accesses, and accesses resulting from erroneous pointer arithmetic. This is done by keeping track of attributes of the referent of each pointer by transforming C code to C++ code, and taking advantage of operator overloading to perform appropriate checks whenever certain operators are applied. Purify[1] detects errors similar to those found by Safe-C, and, in addition, identifies uninitialized memory reads and memory leaks. Purify performs these checks by instrumenting object files and modifying the layout of heap-allocated memory in order to catch access errors. Our approach catches all of these errors in addition to run-time type violations that are not covered by Purify and Safe-C. Furthermore, the warning messages provided by our tool provide a history of suspicious type propagation that can aid in pinpointing the true cause of an error.

In the realm of security, tools have been developed to prevent "stack smashing" (where the return address in the activation record is modified by a malicious agent to obtain control of the program)[10, 11]. Our tool also detects such attacks, which fall under the general category of "type errors" detected by our tool.

A technique to enable efficient checking of array-access and pointer-dereference errors in a multiprocessor environment was presented in [3]. They achieve low-cost checking by creating a version of the program that contains only computations that affect pointer and array accesses, instrumenting that version, and running it in parallel with the original program. We may be able to use this technique to improve our tool's performance.

There have also been a number of efforts to address the problem of identifying errors in C programs due to out-of-bounds array indexes and misuses of type casts based on the use of static analysis. Work on static analysis that can be applied to checking for out-of-bounds array accesses includes [12–16]. Algorithms for points-to analysis that distinguish among fields of structures [17, 18] and for so-called "physical type checking" [19] can also be used to perform static safety checks. However, most of the work based on static analysis cited above has used flow-insensitive techniques, which is likely to cause an enormous number of warnings of possible misuses to be generated when applied to safety checking of real-life C programs. The advantage of a dynamic type-checking tool like the one reported in this paper is the ability to obtain more accurate information about type misuses and access errors, albeit only for ones that occur during a given run of the program.

References

1. R. Hasting and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter Usenix Conference*, 1992.
2. T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.
3. H. Patil and C. Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software–Practice and Experience*, 27(27):87–110, 1997.
4. M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *Proc. of ESEC/FSE '99: Seventh European Softw. Eng. Conf. and Seventh ACM SIGSOFT Symp. on the Found. of Softw. Eng.*, pages 180–198, September 1999.
5. Ckit. http://cm.bell-labs.com/cm/cs/what/ smlnj/doc/ckit/.
6. R. Stallman and R. Pesch. *Using GDB: A Guide to the GNU Source-Level Debugger.* July 1991.
7. B. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical report, University of Wisconsin-Madison, 1995.
8. M. Fagan. Soft typing: An approach to type checking for dynamically typed languages. Technical Report TR92-184, Department of Comp. Sci., Rice Univ., Houston, TX, USA, March 1998.
9. A. Wright. Practical soft typing. Technical Report TR94-236, Department of Comp. Sci., Rice Univ., Houston, TX, USA, April 1998.
10. Immunix stack guard. http://immunix.org/stackguard.html.
11. Stack shield. http://www.angelfire.com/sk/stackshield/info.html.
12. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conf. Rec. of the Fifth annual ACM Symp. on Princ. of Prog. Lang.*, pages 84–96. ACM, January 1978.
13. C. Verbrugge, P. Co, and L.J. Hendren. Generalized constant propagation: A study in C. In *6th Int. Conf. on Compiler Construction*, volume 1060 of *Lec. Notes in Comp. Sci.*, pages 74–90. Springer, April 1996.
14. R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 182–195, New York, NY, 2000. ACM Press.
15. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 321–333, New York, NY, 2000. ACM Press.
16. D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, San Diego, CA, February 2000.
17. B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *6th Int. Conf. on Compiler Construction*, volume 1060 of *Lec. Notes in Comp. Sci.*, pages 136–150. Springer, April 1996.
18. S. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 91–103, May 1999.
19. S. Chandra and T. Reps. Physical type checking for C. In *Proc. of PASTE '99: SIGPLAN-SIGSOFT Workshop on Program Analysis for Softw. Tools and Eng.*, pages 66–75, New York, NY, 1999. ACM.