Chapter 9

Runtime Type Checker (RTC)

The MSE uses tagged memory to enforce memory safety, which is one of several properties that are mandated but not enforced by the C language. In this chapter, we extend the tagged memory approach to check another property that is not enforced by the C language: type safety. The approach, called the Runtime Type Checker (RTC), tags each byte of memory with four bits that encode the runtime type of the value in that byte. Whenever the value in a location is used, the runtime type encoded in the location’s tag is checked against the expected type of the use. If there is a type mismatch, a type-safety error is reported. Assignments are treated specially: if the runtime type of a value being assigned does not match the expected type of the assignment, a type-safety warning is reported to indicate “suspicious type behavior” that may not be a true error. These warnings are often useful for tracking down the root cause of a subsequent error.

The RTC checks subsume the MSE checks, in that memory-safety violations that are detected by the MSE are also detected by the RTC. The RTC checks also subsume checks performed by the debugging tool Purify [Has+92], which uses tagged memory to check for both memory-safety violations and uninitialized memory accesses. However, the additional type information maintained by the RTC allows it to detect more subtle errors that would not be detected by the MSE or Purify — errors that manifest themselves as type-safety violations but that do not violate memory safety or access uninitialized memory.

The RTC is intended for use during program development or testing, which means a relatively high runtime overhead and a few false positives can be tolerated — though it is
important to minimize both. The core type-safety model used by the RTC, described later in this chapter, is chosen to minimize the number of false positives reported while maximizing the RTC's ability to find real errors. In the next chapter, several static analyses are described to improve the runtime overhead of the RTC.

9.1 Motivating Examples

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

9.1.1 Bad Union Access

A very simple example of a logical error that manifests itself as a bad runtime 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:

1. union U {
2.   int u1;
3.   int *u2;
4. } u;
5. int *p;
6. u.u1 = 34;  /* write into u.u1 */
7. p = u.u2;   /* read from u.u2 — warning! */
8. *p = 0;     /* bad pointer dereference — error! */

In this example, an integer value is written into location u (at line 6), and is subsequently read as a pointer (at line 7). The value that is read from u is stored in variable p, which is then dereferenced (on line 8). The symptom of the error is the attempt to use the value 34 as an address on line 8; however, the actual point of the error can be said to be on line 7, when a value of one type is read as if it were another type (i.e., the runtime type of u.u2 is not the same as its static type).
A tool that checks for memory-safety violations, like MSE or Purify, would report an error when line 8 is executed; however, it would not be able to point to line 7 as the source of the error, and it might not report an error at all if the value 34 happened to be a valid memory address.

The RTC would tag the single location corresponding to both u.u1 and u.u2 with the type of the value stored at that location. After the assignment u.u1 = 34 on line 6, that tag would be set to int. At line 7, the int-tagged value is read and assigned via a pointer-typed assignment. This is a type mismatch; therefore the RTC would produce a warning message at line 7 indicating suspicious type behavior. The dereference at line 8 is a use that expects a pointer-typed value. Since the value in p is still tagged int, the RTC would generate an error message indicating a bad type use.

9.1.2 Custom Allocator

C programmers sometimes try to improve the runtime performance of memory management by writing their own custom allocator to use in place of malloc and free. For example, a programmer might allocate a large chunk of memory using a single call to malloc via an assignment like the following:

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

(where BLOCKSIZE is some large integer value). Subsequently, when new memory is needed, a call is made to a user-defined allocation function, myMalloc, which returns a pointer to an appropriate part of the myMemory block. Similarly, calls to free are replaced by calls to myFree, which update appropriate data structures to keep track of which parts of myMemory are currently in use.

Consider the following code fragment:

1. char * cp = (int *) myMalloc(64 * sizeof(char));
2. int * ip = (int *) myMalloc(64 * sizeof(int));
3. cp[64] = 'x';
4. ip[0]++;

...
The custom allocator is used to allocate two arrays, which are both part of the same `myMalloc` block. Suppose the two arrays happen to be contiguous in memory. The dereference `cp[64]` at line 3, which is outside the bounds of the `cp` array, would erroneously write into the first byte of the `ip` array. This would corrupt the `int` value stored at `ip[0]`, and would likely cause the program to produce an incorrect output.

Because both the `cp` and `ip` arrays are part of the same `myMalloc` block of memory, the erroneous dereference `cp[64]` is within the bounds of its intended target, so it does not violate memory safety. This means that memory-safety checking approaches, including MSE, Purify, and the fat-pointer approaches, would not detect the error.

With the RTC, the assignment at line 3 causes the byte of memory at `cp[64]` to be tagged `char`. The increment operation at line 4 is a use of the value in `ip[0]` that expects an `int`-typed value, so the tags of the bytes of `ip[0]` are checked: in this case the tag of the first byte would contain a `char` tag, which does not match the expected `int` type; thus, an error message would be issued to indicate a bad type use.

### 9.1.3 Simulating Inheritance with Structures

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 [Sif++99]. For example, the following declarations might be used to simulate the declaration of a superclass `Base` and a subclass `Sub`:

```c
struct Base { 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:

```c
void f ( struct Base * p ) {
    p->a1 = ... 
    p->a2 = ... 
}
```

and the function might be called with an actual argument either of type `struct Base *` or `struct Sub *`:
struct Base base;
struct Sub sub;
f(&base);
f(&sub);

The C language 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 runtime 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 subclass of struct Base, and while making changes to the code might change the type of one of the common fields, add a new field to struct Base without adding the same field to struct Sub, or add a new field to struct Sub before field b2. For example, suppose a new int field, i1 is added to struct Sub:

    struct Sub { int b1; int i1; int *b2; char b3; };  

Now, when the second call to f is executed, the assignment b->a2 = ... would write into the i1 field of sub rather than the b2 field. The fact that the b2 field is not correctly set by the call to f, or that the i1 field is overwritten with an unintended value, will probably either lead to a runtime error later in the execution, or cause the program to produce incorrect output.

The tracking of runtime types performed by the RTC tool can detect this error. The assignment p->a2 = ... causes sub.i1 to be tagged with type pointer. A later use of sub.i1 in a context that expects an int would result in an error message due to the mismatch between the expected type (int) and the runtime type (pointer).
Note that the erroneous assignment does not violate memory safety, since \( p->a2 \) remains within the bounds of the outermost object \( \text{sub} \). Therefore, memory-safety checking approaches would not detect this error.

### 9.2 Type Safety

Type systems [Car97] are useful for organizing data into conceptual categories, called types, and for describing operations that are permitted on data of a given type. A good type system can improve the quality of code, make programs easier to understand and maintain, and ensure that certain classes of errors do not occur. However, a type system that is too strong can limit the expressiveness or flexibility of the language, and make it difficult to implement certain low-level tasks efficiently.

The C language and its static type system can be described as “strongly typed, modulo some loopholes”. That is, while most of the language can be statically type-checked to guarantee that type errors do not occur at runtime, this guarantee is broken by a few loopholes in the language. These loopholes are necessary to support efficient implementation and flexibility — features that are important to C programmers — and include the following:

1. Array indexing and pointer arithmetic, which may allow a dereference to access memory outside the bounds of its intended target (causing a spatial access error).

2. Storing a pointer to stack or heap objects, which allows the pointer to be dereferenced after the object is deallocated (causing a temporal access error).

3. Reading from a memory location before initializing it, so that the value read may be stale data of arbitrary type.

4. Type casting and union types, which allow any memory location to be used to store values of an arbitrary type, making it possible for an operation that expects one type to use a value of a different type.
The first two loopholes make C programs prone to memory-safety errors, while the third gives rise to uninitialized-memory-access errors — both of which can be regarded as special kinds of type errors. The fourth loophole, type casting and union types, can cause a type error to occur without violating memory safety or accessing uninitialized memory.

9.2.1 C Language and Static Type System

In this section, we review some relevant aspects of the C language and static type system, and establish some simplifying assumptions.

An *lvalue expression* is an expression that refers to a memory location. In the C language, *lvalue* expressions include variable identifiers (x), indirection expressions (*e₁ or e₁->mem), array index expressions (e₁[e₂]), and field selectors e . mem where e is also an *lvalue* expression. The two contexts in C that require an *lvalue* expression are the operand of the address-of operator (&), and the destination of an assignment (the left-hand-side of the operators =, +=, *=, etc., or the sole operand of the prefix and postfix operators ++ and --; we shall collectively refer to these as *assignment operators*). Note that an expression having an *lvalue* does not imply that it refers to a *valid* memory location; instead, it means simply that the expression is treated (by the language semantics) as referring to a memory location.

In a C program, every expression and operator has a well-defined static type. Figure 9.1 shows a simple example program, and lists the static types of the operators occurring in the statement at line 4. Each syntactic occurrence of an operator corresponds to a semantic operation whose behavior conforms to the operator’s static type. For an operator with type τ₁ → τ₂, we say the corresponding operation *uses* its (first) operand with expected type τ₁, and *produces* a value of type τ₂. A predicate test (in an if statement, while loop, etc.) is also considered a *use* of the predicate expression. The comma operator (,) is not considered a use of its first operand.

---

1 We use the notation τₖ to represent a τ-typed *lvalue*, or a reference to a memory location with static type τ. For example, the assignment in Figure 9.1 expects its left-hand-side argument to be a memory location with static type float.
We treat function argument passing and the return statement as assignment operations. We further assume that the assignment operation requires its source and destination operands to have matching static types (modulo type qualifiers); i.e., the = operator is assumed to have a static type of the form $\tau\& \times \tau \rightarrow \tau$. An assignment that does not satisfy this condition can be converted to one that does by adding an explicit cast to the source operand. For example, if $i$ is of type `int`, the assignment $i = 'c'$ can be converted to $i = (\text{int}) 'c'$.

### 9.2.2 Runtime Type-Safety Model

Because C does not include provisions for type-checking at runtime, we must define our own runtime type-safety model for the RTC. This type-safety model should extend C’s static type system naturally, so that real errors that arise due to inappropriate uses of types are recognized as type-safety violations. On the other hand, the type-safety model should not be too restrictive. We want to avoid classifying legitimate operations as type-safety violations.

We first define the *runtime type* of a C expression as follows:

1. a. The runtime type of an *lvalue* expression $e$ is the runtime type of the value last written into the location to which $e$ refers.

   b. If no value has been written into that location, its runtime type is a special uninit type.
c. If the location is not part of allocated memory, its runtime type is a special \texttt{unalloc} type.

2. The runtime type of a non-\textit{lvalue} expression is the type of the value \textit{produced} by that expression’s semantic operation (i.e., the expression’s static type).

We can define a runtime type-safety model that naturally extends C’s static type system with the following rule:

3. For each \textit{use} of an expression $e$ that expects a type $\tau$, the runtime type of $e$ should be compatible with $\tau$.

Assignments require special attention, as they can be modeled in several ways. The RTC treats assignments in the following way:

4. a. An assignment is treated as a generic memory-copy operation that does not change the runtime type of the assigned value.

b. An assignment is not considered a \textit{use} of its destination operand, and the type of the value it produces is the runtime type of the assigned value rather than the static type of the assignment.

c. Additionally, if the runtime type of the copied value is not compatible with the expected type of the assignment, it is considered a type-safety \textit{warning} rather than an error.

This treatment of assignments, which effectively ignores the declared type of the destination location, distinguishes the RTC from related approaches like Hobbes [Bur+03] and CCured [Nec+02]. In Hobbes, a type mismatch with the declared type of the destination location generates a warning message, while in CCured, the runtime type of the value is converted to the static type of the assignment. Legitimate programs that assign into locations declared with a different type — such as by an implementation of a custom memory allocator — would trigger a false positive in Hobbes and be restricted in behavior by CCured.
We relax our runtime type-safety model further with the following conditions that were found to reduce the number of false positives in practice:

5. The address-of operator (&) is not considered a use of its operand; i.e., taking the address of a location does not require the runtime type of the location to be well-typed.

6. All pointer types are treated as equivalent, and type qualifiers (e.g., const, unsigned) are ignored.

7. The literal value 0 (or NULL, or '\0') is considered to be compatible with any scalar type.

8. A use that expects an aggregate type (a structure, union, or array) does not require the runtime type to be compatible with the expected type. In other words, our type-safety model only describes requirements for scalar types.

9. A type cast that does not change the underlying data representation, such as a cast between a pointer and an integer of the same size, does not require the type of the operand to be compatible with its expected type, and has the same runtime type as its operand. We call such a cast a copy cast. A cast that changes the underlying representation, such as a cast between an int and a float, is called a conversion cast. A conversion cast (τ) e is considered a use of e which expects the static type of e, and produces a value of type τ.

To illustrate the treatment of copy and conversion casts, consider the example in Figure 9.2. The single location u is declared as a union that can be accessed as an integer (via u.i), pointer (via u.p), or float (via u.f). We assume that a pointer and an int are of the same size. At line 1, the cast from a pointer to an int is considered a copy cast because it does not change the underlying data bits; this means the value written into u can be legitimately used as a pointer, such as by the dereference in line 2. In our type-safety model, the
union {
  int i;
  char *p;
  float f;
} u;
1. u.i = (int) "xy"; /* copy cast */
2. putchar(*u.p);
3. u.i = (int) 6.7; /* conversion cast */
4. u.f = 6.7;
5. u.i++;

Figure 9.2 Copy and Conversion Cast

value stored in u at line 1 would have runtime type pointer, which is compatible with the expected type of the use at line 2.

The assignments at lines 3 and 4 both appear to assign the value 6.7 to the location u, but they are very different semantically. At line 3, there is a conversion cast from float to int which converts the value 6.7 to the integer 6, and assigns it to the location u; this conversion operation is therefore treated as a use of 6.7 that expects a float-typed value and produces an int-typed value. At line 4, on the other hand, the bit representation of the float value 6.7 is copied directly into u; a subsequent attempt to use this value as an int (e.g., at line 5) would encounter a value that corresponds to the float encoding of the value 6.7, which would probably not be the intent of the programmer. In our type-safety model, the value stored in u at line 4 would have runtime type float, so that the use at line 5 which expects an int-typed value would be a type-safety violation.

9.3 Tracking Runtime Types

The runtime types in the RTC type system consist of the following:

- char, short, int, long, longlong (collectively called INTEGRAL types).
- float, double (collectively called REAL types).
- **pointer**, representing all pointer types.

- **init**, representing initialized data that is compatible with any type. The value zero has type **init**.

- **uninit**, representing uninitialized data.

- **unalloc**, representing unallocated memory.

For aggregate objects (structures and arrays), the runtime types of the component scalars are tracked independently. Enumerations are treated as **int**s (per the C specification), and **typedefs** are resolved to their underlying basic types.

At runtime, each memory location is associated with a tag that encodes its runtime type. The type tags are stored in a “mirror” of memory, with each byte mapping to a four-bit nibble in the mirror. Of these four bits, the first is a “continuation” bit that encodes the extent of the tag (0 denotes the start of a new tag, 1 denotes a “continuation” nibble), and the other three are “data bits” that encode other information. In the first nibble of a tag, the data bits encode the *type-class* of the type — one of **integral**, **real**, **pointer**, **init**, **uninit**, and **unalloc**. If the type is larger than one byte in size, the data bits of the second nibble encode \( \log_2 \) of the size of the tag. (A tag with a size of 1 can be recognized by checking that the continuation bit of the subsequent nibble is 0.) For objects larger than two bytes, the remaining data bits are currently unused (they could be used to encode information for future enhancements or optimizations).

The tags for some common scalar types are illustrated in Figure 9.3. The continuation bit needs to be set to 1 for each nibble in the entire extent of the tag, so that an access to the middle of a tag can be recognized. In fact, the encoding of the type’s size in the second nibble of the tag is redundant, as the size of the tag can be determined by counting the distance to the next nibble with a 0 continuation bit. The second nibble encoding of the size is an optimization that enables the checking of the equality of two tags (of size greater than one) to be done quickly by comparing only the first byte (two nibbles) of the tags.
<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Tag Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>integral 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>uninitialized byte 0</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>integral 0 \log_2 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unused 1 unused 1</td>
</tr>
<tr>
<td>pointer</td>
<td>4 bytes</td>
<td>pointer 0 \log_2 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unused 1 unused 1</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>real 0 \log_2 8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unused 1 unused 1</td>
</tr>
</tbody>
</table>

Figure 9.3 Tag Representation

Note that this tag representation captures implementation-dependent type identities; e.g., if int and long are the same size in a given implementation, their tags would be identical.

The mirror is allocated in \( \frac{1}{2} \)MB pages (each mapping to 1MB of user memory). Pages are allocated on demand, and pointers to these pages are stored in a table indexed by the most significant 12 bits of the user-space address (in a 32-bit address space). Thus, looking up the tag of any given object involves a constant-time lookup in the page table followed by a constant-time lookup in the mirror page.

### 9.4 Instrumentation

The RTC instruments a program via a source-to-source transformation. Working at the source level gives the tool access to the static type information it needs. Programs are instrumented to perform the following actions at runtime:

- Initially, all memory is implicitly tagged with the unalloc type (encoded with the value 0).

- Prior to the top-level call to `main`, each global variable `v` is tagged as follows:
• If \( v \) is initialized to a non-zero value, then \( v \) is tagged with its statically-declared type.

• Otherwise, \( v \) is tagged with the \texttt{init} type. This is consistent with the C requirement that all globals be zero-initialized.

• When a local automatic variable is declared on the stack, its tag is set to \texttt{uninit}. When exiting a function, the tags of all stack variables declared in that function are set to \texttt{unalloc}.

  (Local static variables are initialized like globals, except the initialization occurs the first time the variable’s scope is entered at runtime.)

• At each \textit{use} of an \textit{lvalue} expression \( e \) that expects a static type \( \tau \), the tag of the location \( e \) is looked up in the mirror. If the tag is not compatible with \( \tau \), i.e., if it is neither \( \tau \) nor the \texttt{init} type, then a type-safety error is reported. To avoid cascading error messages, the tag of \( e \) is set to \( \tau \) after an error is reported.

• At an assignment \( e_1 = e_2 \) that expects a scalar type \( \tau \),

  1. The runtime type of \( e_2 \) is determined.

  2. If the runtime type of \( e_2 \) is not compatible with \( \tau \), a \textit{warning} is reported, to signify suspicious type behavior.

  3. If \( e_1 \) is a dereference, the mirror of the location \( e_1 \) is checked: if any of it is tagged \texttt{unalloc}, a \textit{memory-safety error} is reported.

  4. The tag of the location \( e_1 \) is set to the runtime type of \( e_2 \).

• At an assignment \( e_1 = e_2 \) that expects an aggregate type \( \tau \) (where \( \tau \) is a structure, or a union containing a structure),

  1. If \( e_1 \) is a dereference, the mirror of the location \( e_1 \) is checked: if any of it is tagged \texttt{unalloc}, a \textit{memory-safety error} is reported.
2. If $e_2$ is a dereference, the mirror of the location $e_2$ is checked: if any of it is tagged `unalloc`, a memory-safety error is reported.

3. The tags from the location $e_2$ are copied into the mirror of location $e_1$.

The main difference with scalar-typed assignments is that the tags of $e_2$’s components are not checked against the components of the expected aggregate type $\tau$.

- At a function callsite, the tags associated with the actual arguments are stored in a temporary data structure. At the entry of a function, the tags of the actuals are retrieved and assigned to the formal argument variables.
  At a `return` statement, the tag of the return value (if any) is stored in the same temporary data structure; this tag is retrieved at the callsite and used as the runtime type of the function-call expression.

The interface to the RTC runtime system includes the following procedures:

- **SetTag**$(e, \tau)$: sets the tag of location $e$ to runtime type $\tau$.

- **CopyTag**$(e_1, e_2, n)$: copies the tag(s) for $n$ bytes of memory from the mirror of location $e_2$ to the mirror of location $e_1$.

- **VerifyType_use**$(e, \tau)$: verifies that the tag of location $e$ is compatible with type $\tau$. If the types are not compatible, a type-safety error is reported, and the tag of $e$ is set to $\tau$ (to prevent cascading error messages).

- **VerifyType_assign**$(e, \tau)$: verifies that the tag of location $e$ is compatible with type $\tau$. If the types are not compatible, a type-safety warning is reported; the tag of $e$ is not changed.

- **VerifyDeref**$(e, n)$: for a dereference expression $e$, checks that the first $n$ bytes of memory starting from location $e$ are allocated (i.e., are tagged with anything other than `unalloc`); if not, a memory-safety error is reported.
These procedures are implemented using macros whenever possible to avoid the overhead of
function calls.

Figure 9.4 presents a simple example program, and the RTC instrumentation that would
be added at each line. For each declaration of a local variable (lines 1-3), an RTC call is
added to set the variable’s tag to `uninit`. At line 4, the assignment writes an `int`-typed
value into `i`, so a `SetTag` call is added to set `i`’s tag to `int`. At line 5, a `pointer`-typed
value is written into `p`, so a `SetTag` call is added to set `p`’s tag to `pointer`. The assignment
at line 6 causes four things to happen:

1. a `VerifyType_assign` call is added to check that `i`’s runtime type is compatible with
   `int`; if it is not, a warning message is issued.

2. a `VerifyType_use` call is added to check that `p`’s runtime type is compatible with
   `pointer`; if it is not, a type-safety error message is issued.

3. a `VerifyDeref` call is added to check that `*p` refers to memory that is not tagged
   `unalloc`; if it is, a memory-safety error is reported.

4. a `CopyTag` call is added to copy `sizeof(int)` bytes worth of tags from the mirror of
   `i` to the mirror of the location accessed by `*p` (in this case, the mirror of `f`).

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>RTC Instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>int i;</td>
<td>SetTag(&amp;i, uninit_type);</td>
</tr>
<tr>
<td>2.</td>
<td>float f;</td>
<td>SetTag(&amp;f, uninit_type);</td>
</tr>
<tr>
<td>3.</td>
<td>int *p;</td>
<td>SetTag(&amp;p, uninit_type);</td>
</tr>
<tr>
<td>4.</td>
<td>i = 4;</td>
<td>SetTag(&amp;i, int_type);</td>
</tr>
<tr>
<td>5.</td>
<td>p = (int *) &amp;f;</td>
<td>SetTag(&amp;p, pointer_type);</td>
</tr>
<tr>
<td>6.</td>
<td>*p = i;</td>
<td>VerifyType_assign(&amp;i, int_type); VerifyType_use(&amp;p, pointer_type); VerifyDeref(&amp;*p, sizeof(int)); CopyTag(&amp;*p, &amp;i, sizeof(int));</td>
</tr>
<tr>
<td>7.</td>
<td>i = f + 7;</td>
<td>VerifyType_use(&amp;f, float_type); SetTag(&amp;i, int_type);</td>
</tr>
</tbody>
</table>
Finally, at line 7, the *use* of *f* means a `VerifyTag_use` call is added to check that *f*’s tag is compatible with *float*; in this case, *f* was tagged *int* by the assignment at line 6, so a type-safety error is reported. The + expression produces an *int*-typed value that is assigned into *i*, so a `SetTag` call is added to set *i*’s tag to *int*.

### 9.4.1 Library Functions

Certain library functions have type behavior that must be accounted for by the RTC to detect errors or to prevent the reporting of false positives. These include functions that allocate or deallocate memory (e.g., `malloc` and `free`), functions that copy memory (e.g., `strcpy` and `memcpy`), functions that read input (e.g., `fgets`), and functions that return a pointer to memory that is allocated within the library (e.g., `ctime`). Each call to one of these functions is replaced with a call to a wrapper function that performs the necessary tag manipulations to capture the type behavior of the function. The wrapper functions for `malloc` and its relatives set the tags of an allocated block to *uninit* (or, in the case of `calloc`, to *init*), while the wrapper for `free` sets the tags to *unalloc*. These wrappers also do some bookkeeping to enable `free` to know how many bytes to deallocate.

For a call to an uninstrumented function, the runtime type of the return value (if any) is assumed to be equal to its declared type. This behavior allows instrumented modules to interoperate gracefully with an uninstrumented module, which can be useful if a programmer only wants to debug one small component of a large program. Only if the function affects type behavior in a way that is externally visible (besides in the return value) would a wrapper function need to be written to prevent false positives from being reported.

### 9.5 Experience with Finding Bugs

To test the effectiveness of the RTC as a debugging tool, we used Fuzz [Mil+95] to find Solaris utilities that crash on some random input, and instrumented five such programs for testing (`nroff`, `plot`, `ul`, `units`, `col`). In each case, the RTC reported useful error messages before the program crashed. Furthermore, the tool detected a number of bugs in the Spec 95
(go, jpeg, vortex) and Spec 2000 (mesa, parser) benchmarks. To help determine the root cause of each error, the RTC can be used in conjunction with an interactive debugger like GDB [Sta+02]. When a warning or error message is issued, a signal (SIGUSR1) is sent, which can be intercepted by GDB; the user is then able to use the facilities of GDB to examine values in memory, including the RTC mirror, to track down the cause of an error.

All of the errors we detected were either memory-safety errors or uses of uninitialized variables, i.e., errors that could be detected by Purify. However, in three of the cases (nroff, vortex, and parser) the RTC detected some invalid accesses which happened to land in allocated memory but accessed data of a different type — these invalid accesses would not be detected by Purify. For example, in nroff, an array of pointers is erroneously accessed with a negative index. The retrieved value, when dereferenced, causes a segmentation fault. It turns out the erroneous array access reads a word from an array of characters declared elsewhere in the program. Thus, the erroneous array access causes the RTC to report a type-safety warning, because the tags of the read value (a series of chars) do not match the expected type (pointer). Purify does not detect the invalid array access, and only reports an error when the erroneously-read value is dereferenced. Thus, the RTC warning identifies the point of the program at which the cause of the error (the erroneous array access) occurs; furthermore, if the value read from the erroneous array access happened to land in allocated memory, the subsequent invalid dereference might not be detected by Purify.

We can easily create examples (such as the ones given in Section 9.1) for which the RTC would report errors that are not detected by Purify; however, we have not yet found examples of those kinds of bugs in real programs. We believe such bugs are more likely to arise during the software-development cycle, and we have not had the opportunity to experiment with the RTC in such a setting. The programs we have used to date for testing are in most cases robust code that has been in use for some time.
9.5.1 False Positives

In our experiments, we have found that the RTC reports very few false positives. The cases we encountered of legitimate operations that trigger RTC warnings or errors include the following:

- Byte-wise copying: A function that copies arbitrary data from one location to another is usually implemented such that the copy is performed one byte at a time (using a `char` pointer). Thus, each assignment expects a `char`, which causes the RTC to report a warning if the underlying data is of a different type.

A shortcoming of our current tag encoding scheme is that it does not enable us to encode pieces of a large tag and faithfully reconstruct the full tag from the pieces. Thus, if the program copies an `int` value (a four-byte INTEGRAL tag) one byte at a time, our tag-manipulation routine would produce four successive `char` tags (four one-byte INTEGRAL tags).

- Data overlays: In programs that are interested in the machine representation of data, such as in systems or networking, the same series of bytes may be operated on using different data overlays. A common example in networking is the IP address, which may be treated as an array of four bytes, or as a single 32-bit word. If the data is written as one type and used in a context that expects a different type, an RTC error would be reported.

- Hash computation: When computing a hash value for a pointer, the `pointer`-typed value (which has been `copy-cast` to an `int`) is used by an arithmetic operation that expects an `int`-typed value, which triggers the RTC to report a type-safety error.

With the diagnostic messages reported by the RTC (which include the file name and line number, the offending C expression, and the RTC tags), it is usually quite straightforward to recognize an error message as a false positive. The RTC includes a facility to suppress output of duplicate messages (messages of the same kind originating from the same file and line number), or to filter out certain error messages completely.
Figure 9.5 gives the runtime overhead of the RTC, as the ratio of the running time of the RTC-instrumented executable over the running time of the original (uninstrumented) executable. The average is $44 \times$ slowdown, which is quite high, but acceptable in a debugging setting.

Figure 9.6 gives the slowdown in compilation time, comparing the time it takes to instrument and compile a program with RTC to the time it takes to compile the original uninstrumented program. Recall that the RTC is a source-to-source transformation, and the instrumented procedures are implemented as macros whenever possible; thus, the instrumented source file is considerably larger and more complicated (with nested expressions, etc.). Nonetheless, the compilation slowdown (average $17 \times$) is still very reasonable; more importantly, the approach scales to large programs.
Figure 9.6 Compilation-Time Slowdown

(average = 17.4x)