A C/C++ Front End for the Daikon Dynamic Invariant Detection System

by

Benjamin Morse

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by ....
Michael D. Ernst
Assistant Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A C/C++ Front End for the Daikon Dynamic Invariant Detection System

by

Benjamin Morse

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Daikon dynamic invariant detection suite is a system designed to extract specifications from programs, in the form of information about their variables and their relationships to each other. It does this by instrumenting the source code of a target program, which inserts code that directs the program to output the values of its variables and other information when run. This data is then sent to Daikon proper, which performs analysis on it and reports invariants about the program variables. Daikon is a useful tool that can suggest invariants beyond those provable by current static methods.

While the invariant analysis tool is language independent, the front ends — tools that instrument of the user code — must be written for every language to be instrumented. There is a huge base of pre-existing code written in C/C++ for which invariants can be discovered. C/C++ are also widely deployed, comprise a large segment of software currently in development, and are therefore valuable candidates for analysis. The key difficulty in instrumenting a type-unsafe language like C is that the instrumented program has to determine what variables are valid, and to what extent, so that it does not output garbage values or cause a segmentation fault by dereferencing an invalid pointer. This thesis details the implementation and performance of a Daikon front end for the C and C++ languages.

Thesis Supervisor: Michael D. Ernst
Title: Assistant Professor

Acknowledgments

I owe great thanks to Michael Ernst, for advising me and providing me with technical and emotional support, as well as creating the Daikon system of which `dfec` is a part. I am also grateful to the rest of the Program Analysis Group at MIT LCS, in particular Michael Harder for providing invaluable help in integrating `dfec` with the Lackwit type comparability system. Emmanuel Renieris gave me very useful design input, without which in-place struct construction (and therefore struct instrumentation) would have been impossible.

I am eternally indebted to Chris Goodwin, Thomas Fini Hansen, Nicole Brunet, Michelle Sakayama, and Cheryl Urbani for pushing me, encouraging me, and keeping me stable through the duration of the research involved in this project.

Mom and Dad, thanks for the love and financial support.

This research was supported in part by NTT, Raytheon, an NSF ITR grant, and a gift from Edison Design Group.

Contents

List of Figures

Chapter 1

Introduction

Invariants are statements about a program that describe relationships among states of a program. For instance, they can be used to describe the changes in a variable between an entry and an exit of a procedure, or to state a mathematical relationship between two variables that always holds true, over the entire run of a program.

Static analysis techniques can be used to discover some invariants. However, a sound static analysis is inherently conservative, as it can only output invariants that are rigorously provable. In addition, these techniques are frequently so complex that they are limited by what analysis they can complete in a reasonable time bound. Proofs about interesting properties of the program must be built up from small proofs, some of which cannot be statically verified because of unavailable code (for instance, calls to a compiled external library.) As a consequence, static analysis techniques must often be supplied with a programmer-written specification guaranteeing certain external conditions to be fulfilled.

In contrast, dynamic analysis needs to be told nothing about the source code or context of the target program, as it relies only on the data it receives from runs of the program. The source code is typically modified to generate this data for analysis, but the analysis step itself does not need to have the source available. Given a sufficiently complete test suite, dynamic analysis can suggest potential invariants about a program that a static analysis package would have no hope of finding, because they cannot be mathematically proved. The Daikon dynamic invariant detector [Ern00] is a language-independent system that can perform this dynamic analysis, when combined with a language-specific front end.

C/C++ is the language of choice for many developers due to a large installed base, extensive portability, and the availability of myriad development tools. In addition, the reference implementation of many algorithms is written in C/C++. This yields a large body of work in which potential invariants could be discovered, and suggests that a tool for dynamic analysis of C/C++ would be useful. Dynamic invariant detection is also useful to maintainers of pre-existing, poorly documented code. Dynamic analysis using existing test suites can pick up properties of the code that will help a maintainer understand the way the code is written. As C++ has been around about 20 years [Str00], and C has been around about 25 [KR88], there exists a large body of code that could benefit from the existence of a C/C++ Daikon front end.

However, developing a C/C++ front end is a serious technical challenge. The data collection required for dynamic analysis involves accessing variables at times when the programmer is not specifically referencing them. In a type-unsafe language like C, it is hard to determine what data is accessible, and to what extent the data can be accessed without performing an invalid access. Accesses to uninitialized memory that is available to the programmer, but known to the programmer to not be useful, could cause garbage to be output and diminish the accuracy of dynamic analysis. It is even possible that a spurious access could cause the running program to crash, by dereferencing a pointer that contains an uninitialized value, or by accessing far beyond the bounds of a valid array. This thesis describes a system that performs C/C++ code transformation to yield a new, "instrumented" version of the code that can be run to generate data for dynamic invariant analysis. The system also contains a runtime library that uses various methods of accounting to determine what data is accessible, useful, and valid, in order to prevent output of uninitialized values or causing segmentation faults.

Chapter 2 discusses the overall design of the system and the motivations for its components. Chapter 3 describes the implementation, as well as implementation challenges, the solutions that were developed to solve them, and features that were added to make the system more flexible and useful to users. Chapter 4 contains the testing methodology used to analyze the correctness and effectiveness of the system, as well as descriptions of tools developed for these tests and their use. Chapter 5 describes possible future extensions to the system, and Chapter 7 reviews the system design, and makes conclusions about what was learned over the scope of this project.

Chapter 2

Technical Approach

Daikon analyzes properties that can be observed over a run of a target program. There are three main steps in this process:

1. Instrumentation via `dfec`
2. Trace file generation
3. Trace file analysis

First the target program must be processed by `dfec`, the executable component of the Daikon front end for C/C++. `dfec` does two things. It creates a file referred to as the `decls` file, which contains type and scope information about the variables in the program for later use by the invariant detector. In parallel, it transforms the input source, adding code to the target program. This whole process is called instrumentation.

The second step, trace file generation, consists of running the instrumented program. The execution of the instrumented program parallels the execution of the

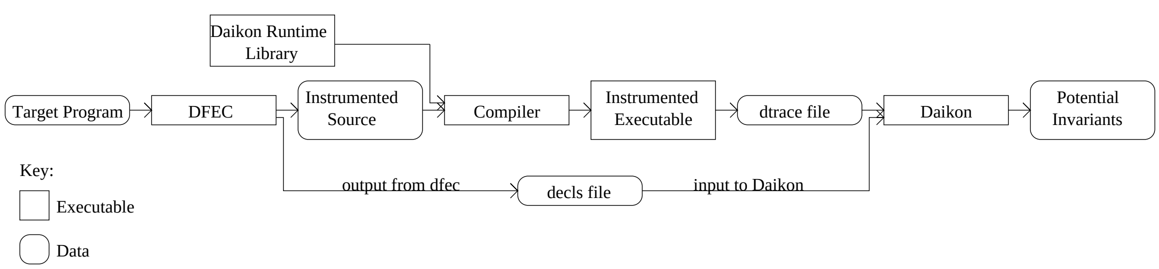

Figure 2-1: The full process of dynamic analysis over a C/C++ program.

original program, except that the code added by `dfec` is called at specific points in the program. The function of the added code is to output, at selected program points in the scope of the transform (typically, the entry and exit points of each function, hereafter referred to as PPTs), the values of all variables visible from that scope. The added code must not affect the input-output behavior of the target program in any other way. The instrumented program is compiled, and run through a test suite, similarly to the way a program would be tested for coverage, profiling, or bug detection, exercising the code thoroughly to get as much and as varied data as possible. During the run of the program, the code added by the transformation outputs data regarding variable state to a trace file, also called the `dtrace` file.

Finally, once the run is complete and the `dtrace` file has been created, Daikon is given the `decls` and `dtrace` files produced by `dfec` and the instrumented program respectively. It examines the values, looking for relationships it can express as potential invariants. The process of instrumentation, compilation, `dtrace` generation, and invariant detection is shown in Figure 2-1.

Daikon takes two files as input: a `decls` file and a `dtrace` file. A toy program and its corresponding `decls` file and `dtrace` file are shown in Figure 2-2. `dfec` produces the `decls` file at the time of instrumentation. It contains declarations of all the different PPTs in the source, and lists what variables will be output at each instance of that program point. For each variable, it lists the declared type (the language-specific type of the variable as it was declared in the target program), the representation type (a language-independent type that Daikon uses internally to represent program data), and the comparability type (an identifying string used to separate variables into comparability classes, described in Section 3.9.2). Daikon uses this data to determine what type of invariants should be looked for, and among which variables to calculate relationships.

The instrumented program is then compiled and run, and produces the `dtrace` file. At each PPT that the program executes, it outputs all the variables that were declared in the `decls` file to the `dtrace` file.

For each variable in the dtrace file, the name is listed, then the value, then a "modbit" to indicate the status of the variable at the time of output. A modbit of 0 means it was valid, but unchanged since last output (this is not implemented in `dfec`, however). A modbit of 1 means it was valid. A modbit of 2 means it was invalid, and should not be included as a sample during invariant detection.

The C/C++ front end consists of two parts: the `dfec` executable and the Daikon runtime library. The `dfec` executable is a source-to-source transformation program that takes in a target program, and outputs an instrumented version of the program. The Daikon runtime library ("DRT") implements the functions and classes used by the code added in the instrumentation step, such as the functions and classes necessary to track and output the instrumented program's variables.

2.1 `dfec` Executable

The `dfec` executable takes in source code and outputs both instrumented source and a `decls` file. Daikon uses the `decls` file to parse what appears in the `dtrace` file, so the set of variables output by the DRT at a given program point must match the set of variables declared at the same program point in the `decls` file. The `dfec` executable is based on the EDG C/C++ front end [EDG00]. This tool performs preprocessing on C/C++, builds an IL (intermediate language) tree of the statements in the target program, then unparses them again, yielding an output semantically identical to the input (after macro preprocessing). By modifying the back end, the unparsing step, code transformations are applied.

`dfec`'s primary code transformation is adding output calls to the source at specific program points. The program points chosen for instrumentation are the entry and exit points of each function. This allows Daikon to recapture preconditions and postconditions for functions, which is how traditional invariants are stated. When generating the text for an entry to or exit from a function or class method, `dfec` examines the IL tree for variables in scope. For each one, it adds an entry to the `decls` file, and outputs a call to a handler function (defined within the DRT) to the source. These handler functions simply output variable names, values, and modbits

(trivial.c)

```
int foo(int arg) {
  return arg+1;
}

int main() {
  int x = 3;
  x = foo(x);
  x = foo(x);
  return x;
}
```

(trivial.decls)

```
DECLARE
std.foo(int;)int:::ENTER
arg
int
int
1

DECLARE
std.foo(int;)int:::EXIT1
arg
int
int
2
return
int
int
2

DECLARE
std.main()int:::ENTER

DECLARE
std.main()int:::EXIT2
return
int
int
3
```

(trivial.dtrace)

```
std.main()int:::ENTER

std.foo(int;)int:::ENTER
arg
3
1

std.foo(int;)int:::EXIT1
arg
3
1
return
4
1

std.foo(int;)int:::ENTER
arg
4
1

std.foo(int;)int:::EXIT1
arg
4
1
return
5
1

std.main()int:::EXIT2
return
5
1
```

Figure 2-2: A trivial program and its associated `decls` and `dtrace` files.

```
Original              |  Instrumented
                      |
int foo(int bar) {    |  int foo(int bar) {
  int baz;            |    daikon_output_to_dtrace("std.foo(int;)int:::ENTER\n"),
  baz = bar + 3;      |      daikon_output_int("bar", int(bar)),
  ...                 |      daikon_output_to_dtrace("\n");
                      |    int baz;
                      |    baz = (bar + 3);
                      |    ...
```

Figure 2-3: An example of code inserted at a function entry program point by instrumentation.

to the `dtrace` file. An example of the code inserted during instrumentation appears in Figure 2-3.

A second transformation changes variable types from primitive pointer types to DaikonSmartPointer, a templatized type described in Section 2.2.1. Textually, this just involves changing all type expressions of the form `T *` to an expression of the form `DaikonSmartPointer<T>` .

2.2 Runtime Library

At instrumentation time, all that the `dfec` executable knows about variable validity is what variables are in scope. It cannot determine statically whether or not a pointer has been initialized, or the extent of a valid pointer, as these can change from execution to execution. Therefore, its responsibility is limited to keeping track of the list of variables in scope, and generating calls to output handlers for them at instrumentation time.

At runtime, however, more information about variables can (and must) be determined. The contents of a variable are not necessarily valid, even if it's in scope. The DRT, or Daikon runtime library, contains all the mechanisms for monitoring the status of user variables in-memory (as described later, in Section 3.5) as well as outputting them to the `dtrace`. The instrumentation process connects the statically available information to the runtime-available information by inserting calls to the DRT for each variable that needs to be output.

2.2.1 Smart Pointers

It is infeasible to statically determine size of the contents of a pointer variable, or even whether the pointer is valid, so runtime bookkeeping for pointers is necessary to determine how much data to output, or if data should be output at all. In C/C++, a pointer variable can point to a single valid byte, a block of a thousand bytes, or even (when uninitialized) to an illegal memory region which would cause a segmentation fault if referenced. While the `dfec` executable is capable of statically determining what pointer variables are in scope, it cannot determine their validity or size. Instead, all pointer variables in the target program source are replaced with instances of a template class, DaikonSmartPointer, adding the functionality that all pointer variables monitor the validity and size of their contents at runtime.

A smart pointer is a use of the proxy design pattern [GHJV95] that replaces a regular C/C++ pointer variable [PW00]. It provides the indirection operators (`*` and `->`), and thus acts transparently as a normal pointer variable would, without affecting the behavior of the target program. Smart pointers typically add functionality, such as garbage collection. The smart pointers in the DRT, by contrast, keep track of the extent to which the memory they point at is valid. The structure of the smart pointer contains a `base` field, an `index` field, and a `birthcount` field. The `index` field is used for the indirection operation, and has the same value as the primitive pointer it replaces would have had. The `birthcount` field is a nonce used to keep track of the number of times the block pointed to has been allocated, and is described in more detail in Section 3.5.2.

The `base` field is another pointer that points to the lowest accessible byte in a memory region (like a block returned from `malloc()` or a fixed-length array), and is used to relate smart pointers that point to different offsets in the same memory region. The `base` field starts off being the same value as the `index`. However, when one smart pointer is derived from another by pointer arithmetic, the `base` pointer is passed to the resulting smart pointer unmodified. This allows two smart pointers who both point into the same contiguous area of memory (like differing indices in an array, or pointers into the same string) to share this common `base` field. This `base` field is used as a key to look up memory region information in the DaikonBasemap structure, described in Section 2.2.2.

The array subscript, addition, and subtraction operators are overloaded, like the indirection operators. They keep track of the the contiguous stretch of memory beyond the `base` pointer that has been addressed by the instrumented program, for later reference by the output handlers, so that only memory that has been "touched" will be output. Each of these operators implements the required functionality to remain transparent, but also takes note of the maximum address referenced beyond each base, keeping track of this in the DaikonBasemap structure.

2.2.2 Basemap

We want to associate data validity with the data itself, rather than the pointer that points to it. Any number of pointers can point to the same block of data, and any information we gain about the block through one pointer, we wish to share with all other relevant pointers. Therefore, we need some sort of structure that associates the `base` field of a smart pointer with a structure containing information about the block.

This need is fulfilled by the DaikonBasemap structure (also referred to as "the basemap"). Through the basemap, the `base` field of every smart pointer is mapped (non-uniquely) to an instance of a class called DaikonPtrInfo (see Figures 2-5 and 2-4). These structures contain a reference count (the number of smart pointers that have a base equal to the key), as well as a pointer just past the last "seen" element. (They contain other fields as well, discussed in Section 3.5.1, but these are implementation details.) When a new smart pointer is constructed, it creates a new DaikonPtrInfo struct, setting the refcount to 1, and makes an entry in the basemap associating the smart pointer's `base` field with this new DaikonPtrInfo. With an assignment, subscript, or addition operation, a new smart pointer is created. The new smart pointer looks itself up in the basemap, finds the the DaikonPtrInfo made by its predecessor, and increments the refcount. When each smart pointer is destroyed, the refcount is decremented. When all references are dropped (i.e. when the refcount

```
Original              |  Instrumented
                      |
char *X = "HELLO";    |  DaikonSmartPointer<char> X = "HELLO";
char *Y = X + 2;      |  DaikonSmartPointer<char> Y = X + 2;
char *Z = "WORLD";    |  DaikonSmartPointer<char> Z = "WORLD";
```

Figure 2-4: A code snippet that produces the heap layout shown in Figure 2-5.

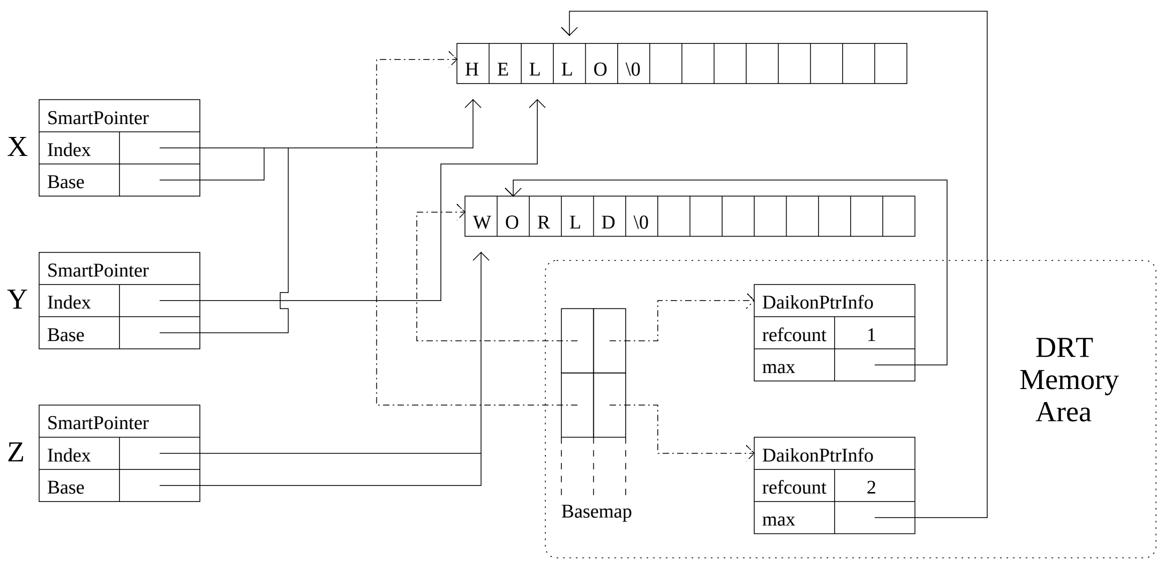

Figure 2-5: Heap layout of the instrumented program of Figure 2-4.

reaches zero), the memory is no longer addressable by the programmer. The DRT knows that if a new smart pointer is constructed to point to that block, it must have come from a new `malloc()` call and that the extent should be reset. Sharing DaikonPtrInfos between smart pointers that have the same base, although differing indexes, allows an offset smart pointer to take advantage of information gained when another pointer, pointing to the same block, updates entries further on in the block, making them valid.

Chapter 3

Implementation

This section discusses, in greater detail, the design decisions that were made during the implementation of `dfec`. Lower level details of smart pointers and the "basemap" mechanism are covered, along with features that were added to enhance usability.

3.1 EDG C++ front end

The static analysis required by instrumentation needs a C/C++ front end including a preprocessor, parser, and semantic checker. Building one from scratch is infeasible for the scope of this project. This being the case, I decided to build `dfec` off of an existing front end.

`dfec` is an extension of the Edison Design Group's C++ front end [EDG00]. The EDG front end performs preprocessing on C++ source, builds an intermediate language tree (referred to in the EDG source as the "IL tree," but also known in compiler terminology as the Abstract Syntax Tree), and then unparses the IL tree back to source. `dfec` consists mostly of additions to the unparsing step, such that when certain expressions are expanded to text (e.g., a procedure body), instrumentation code is added (continuing the same example, output function statements are added before and after the procedure body is output).

The EDG C++ front end conforms to the ANSI C++ standard, accepting ANSI C++ input and producing ANSI C++ output. As many ANSI C programs are also ANSI C++ programs, `dfec` handles a large subset of the ANSI C language (with the notable exception of C programs that have local definitions of tokens that are keywords in the ANSI C++ definition, e.g., `bool`). Many older C programs are written in a dialect of C called K&R (for Kernighan and Ritchie [KR88]), which `dfec` does not handle. However, a program called `protoize` [Gui] is free and widely available which performs conversions from the K&R dialect to ANSI.

3.2 `gcc` integration

Once code is instrumented, it must be compiled and then executed. This requires that a C++ compiler, as well as system library headers, be installed on the user's machine. Because of architecture differences and the widespread practice of proprietary extensions to the C/C++ languages, there are very few (if any) compilers and accompanying system libraries that are written in entirely ANSI C. Given this limitation, I chose to target a specific compiler and set of system library headers, to avoid the massive amount of work involved implementing workarounds for multiple compilers and multiple sets of system library headers. `gcc` version 2.95.3 was chosen as the target compiler because of its widespread availability and the fact that it's free. `dfec` is written to produce instrumented code that can be compiled with `gcc`, and requires in fact that `gcc` be installed, as `dfec` calls `gcc` to determine the system include path.

The EDG front end does full preprocessing (expansion of macros and `#include` directives) of input source, which is necessary for it to construct its IL tree. A set of system library headers is necessary to expand the `#include` directives. System library headers contain function signatures and non-primitive type definitions that are necessary for EDG to correctly parse the source. When run, `dfec` determines the location of the system library headers by executing `gcc -v -E` (for verbosity and preprocessing-only) on an empty source file, and capturing the output. `gcc` outputs the location of the system library headers according to its own configuration files, and `dfec` incorporates this into its own data structures. Then, when an `#include` directive is encountered, `dfec` can find the appropriate system library header to include.

3.3 `libc` instrumentation

Many C/C++ programs are linked to external libraries such as `libc`, which may exist only in binary form. Since `dfec` needs to modify source code to change primitive pointers to smart pointers, binary-only libraries cannot be instrumented. Any operation on memory that occurs inside these libraries cannot be tracked by smart pointers, so without a mechanism that takes into account the semantics of these library functions, information about these memory blocks can become inaccurate.

Instrumented programs still operate correctly when interfacing with non-instrumented libraries. `dfec` transforms all smart pointers into ordinary pointers before passing them as arguments to uninstrumented functions. However, if a pointer is passed to an uninstrumented function, and the function modifies the pointed-to data, the DRT has no way of automatically knowing to what extent the data is now valid. I implemented a workaround for this for certain `libc` functions. `dfec` does source rewriting, replacing calls to `libc` functions with calls to wrapper functions that call the original, then update the basemap according to knowledge about the effects of the function.

3.3.1 `string.h` wrapper functions

For each function from the `libc` header file `string.h`, I wrote a wrapper function that calls the proper function, then updates the basemap with the valid extent of each string argument (and occasionally, the return value). The string functions in particular require this, because almost all of them modify pointed-to data in a way that can change the valid extent (`strcat()`, for example, concatenates strings together, and thus extends the reach of the destination string). Luckily, string functions are also simple to write instrumented wrappers for, because of the knowledge that the string arguments will be zero-terminated. See Figure 3-1 for an example of a wrapped string function.

```
DaikonSmartPointer<char> daikon_feel_out_string(const char *s) {
  if (daikonBasemap[s]) {
    daikonBasemap[s].max_seen =
      max(daikonBasemap[s].max_seen, s + strlen(s) + 1);
  }
}

DaikonSmartPointer<char> DAIKON_strdup (DaikonSmartPointer<char> s) {

    DaikonSmartPointer<char> retval = strdup(s);

    daikon_feel_out_string(s);

    return daikon_feel_out_string(retval);

}
```

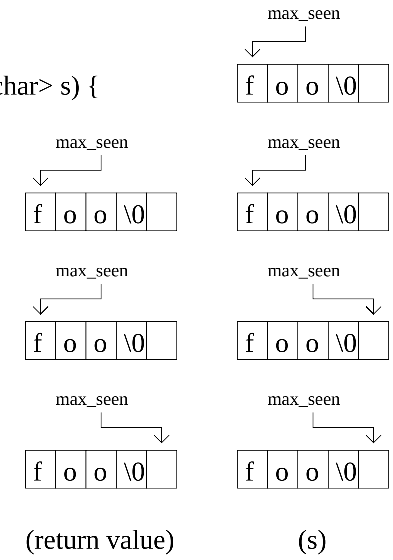

Figure 3-1: Example of a wrapper for the string function `strdup()`.

3.3.2 `malloc()` and `free()` handling

The wrapper functions for `malloc()` and `free()` (`DAIKON_malloc()` and `DAIKON_free()`, respectively) are more involved. In addition to keeping track of the valid extent of heap memory, they must also set up the other fields in the DaikonPtrInfo structure (see Figure 3-3) that keep track of type information and whether or not the constructor has been called. This is all detailed in Section 3.5.

3.4 Smart pointer implementation

DaikonSmartPointer overloads pointer operators, in order to transparently behave like a primitive pointer variable would. Each overloaded operator performs the normal function, as well as recording information in the basemap. An example is given in Figure 3-2.

The example demonstrates the effects of a constructor, `operator []()`, `operator +()`, and `operator *()`.

```
DaikonSmartPointer<char> p1, p2;
char c;
const char *str = "string";

p1 = str;           /* p1.base = p1.index = str;
                       new basemap entry created:
                       daikonBasemap[p1.base] = {
                         modbit = 1,
                         max_seen = str,
                         refcount = 1
                       } */

c = p1[2];          /* p1.operator[](2) returns 'r',
                       updates basemap:
                       daikonBasemap[p1.base].max_seen =
                         max(daikonBasemap[p1.base].max_seen,
                              (p1.index+2)+1); */

p2  = p1 + 1;       /* p1.operator+(1) updates basemap:
                       daikonBasemap[p1.base].max_seen =
                         max(daikonBasemap[p1.base].max_seen,
                              (p1.index+1)+1);
                       p2.base = p1.base; p2.index = p1.index + 1;
                       p2 constructor updates basemap:
                       daikonBasemap[p2.base].refcount++; */

c = *p2;            /* p2.operator*() returns 't', updates basemap:
                       daikonBasemap[p2.base].max_seen =
                         max(daikonBasemap[p2.base].max_seen,
                              p2.index+1); */

p2 = NULL;          /* p2.operator=(NULL) updates basemap:
                       daikonBasemap[p2.base].refcount−−;
                       p2.base = p2.index = NULL; */
```

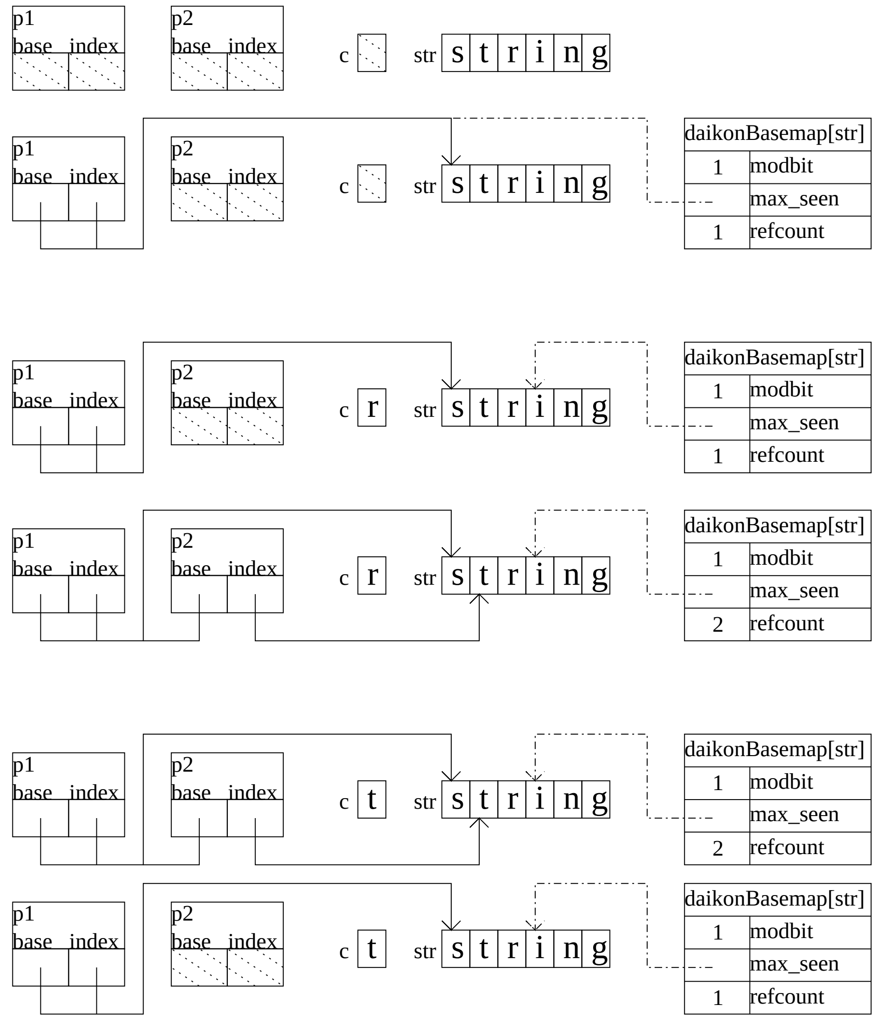

Figure 3-2: Demonstration of smart pointer operator overloading.

3.5 In-place smart pointer construction

The replacement of primitive pointers with smart pointers yields a subtle difference that requires careful management — primitive pointers do not require construction, but smart pointers do, because they are structures, and C++ structures must be constructed in order to initialize their values. Normal pointer variables that are declared in global scope, an argument list, or in the body of a function are stack-based variables. All stack-based pointer variables, once transformed to smart pointers, are set up to be automatically constructed by the compiler. However, in a low-level language like C, the heap is not compiler-managed in such a friendly way. A smart pointer variable that is allocated on the heap will not be automatically constructed by the compiler.

`malloc()` and `free()` are the C functions that allocate and deallocate blocks of memory on the heap. These blocks of memory can store values of any type, unlike in higher level languages like Java, where instances of a type are explicitly constructed as such. In C, there is no such thing as a constructor, so C code generally uses `malloc()` to allocate memory for a struct, and then starts using it.

This leads to a problem in instrumented source when a struct has a field of a pointer type. Under instrumentation, this is transformed to a smart pointer — an object that requires construction. If instrumented code allocates a block of memory and then starts treating it as any struct that has a smart pointer field, it could damage the coherence of the basemap. When a smart pointer is assigned another value, it decrements the refcount of the old address in the basemap, and if it's pointing to a garbage value, it will improperly decrement the refcount for that address. Most likely, this will decrement the refcount of an unaddressed block below zero, although it could possibly decrement the refcount of a valid memory block to zero. This could cause a valid block to be marked as invalid and not be output.

To cope with this, all smart pointers, and structs that contain smart pointers, need to be explicitly constructed on the heap, and destroyed when they're deallocated. This process is performed in three steps — pre-initialization by `DAIKON_malloc()`, construction and type setup by the DaikonSmartPointer class, and destruction and deallocation by `DAIKON_free()`. Sections 3.5.2 through 3.5.4 describe the three steps. However, first we till take a detailed look at the DaikonPtrInfo structure, which is used to keep track of memory state.

3.5.1 A detailed look at DaikonPtrInfo

Until now, I have only discussed the `refcount` and `max_seen` fields of DaikonPtrInfo. Figure 3-3 details all the fields in the DaikonPtrInfo structure.

`max_seen` is, as mentioned before, a pointer to the first byte just beyond the "valid" extent of the block. `refcount` is the number of smart pointers who point into the block (i.e., the number that share the same`base` field). The `modbit` is similar to the modbit described at the beginning of Chapter 2, but is used slightly differently in this context: a `modbit` of 1 means that a memory block is either on the stack, was provided from outside, or is on the heap and has not been invalidated by `DAIKON_free()` yet; and a `modbit` of 2 means that the block was allocated by `DAIKON_malloc()`, but has subsequently been `DAIKON_free()`d.

`bounds` is a pointer that is similar to `max_seen`. It points to the first byte just beyond a memory region, but the semantics are slightly different: where `max_seen` marks off how much of a memory region has been accessed, `bounds` marks off how much of a memory region the DRT should ever allow to be accessed. This is not always known, such as in the case of pointers obtained from system library calls, and in those cases it is set to `NULL`. However, in the case of a fixed-length array or a block returned from `DAIKON_malloc()`, it is known, and is set appropriately. `bounds` is used both to inform the user of illegal accesses by the input program (covered in Section 3.8.1) and to calculate the number of structs that need to be constructed in-place (covered in Section 3.5.3). `leeway` is only relevant when `bounds` is non-`NULL`, and contains the number of bytes on either end of the block that `dfec` put there for array padding. Array padding is described in Section 3.8.2.

`from_malloc` simply denotes whether a block came from a `DAIKON_malloc()` call. When it is false, the remaining fields (`from_malloc`, `unformatted`, `birthcount`,

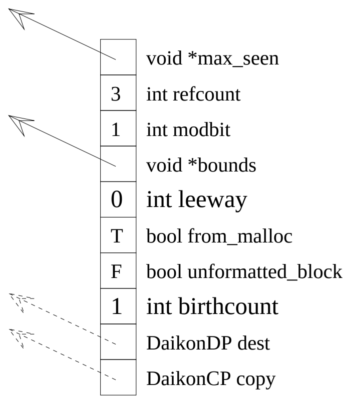

Figure 3-3: The DaikonPtrInfo structure.

`dest`) and `copy` are unset. These fields are only used on blocks that came from `DAIKON_malloc()`. if `from_malloc` is set, then `unformatted` denotes whether it has been constructed in-place (a process described in Section 3.5.3). The `birthcount` field is used to count how many times a particular address has been returned by `DAIKON_malloc()`. This `birthcount` can by compared with the `birthcount` field of a smart pointer to see if the block has been `free()`d and re-`malloc()`ed since the last use of the pointer. This comparison (and the need for it) is discussed in Section 3.5.2.

The remaining two fields are function pointers that are necessary to perform operations that depend on the specific type that the block contains. Once the block has been constructed in-place, `dest` is set to point to a function that `DAIKON_free()` calls to destroy the structs that were constructed. This is described in Section 3.5.4. Similarly, `copy` is set to point to a function that overloaded versions of `memcpy()`, `memmove()`, and `bcopy()` can call to copy a memory region from one location to another, while ensuring constructors are called when copying memory regions that contain structures that may have smart pointer fields.

3.5.2 Role of `DAIKON_malloc()`

Psuedocode for the `malloc()` wrapper function is in Figure 3-4.

When `DAIKON_malloc()` is called, it has no way of knowing what type is going to be stored in the memory region. However, it does know the block's size in bytes. `DAIKON_malloc` first calls `malloc()` itself, and sets up the normal fields in the basemap. In addition, it marks the block as having come from `DAIKON_malloc()` and as "unformatted", as shown in Figure 3-3. Normally, The `refcount` field holds exactly the number of smart pointers that point to the associated block, but in the case of a `DAIKON_malloc()`'ed block, it's incremented by one to represent the `malloc()/free()` pair. This is because when the block has been freshly `malloc()`'ed, it does not yet have anything pointing to it until the pointer assignment, which is where the smart pointer takes over. Adding one to the refcount to represent the `malloc()/free()` pair keeps the DaikonPtrInfo structure from being considered "stale", as DaikonPtrInfos with refcounts of zero are overwritten by the smart pointer constructors.

`birthcount` management

`DAIKON_malloc()` also sets the `birthcount` field. The `birthcount` field is zero for addresses on the stack, but for every address returned by `DAIKON_malloc()`, it is greater than zero. If `DAIKON_malloc()` returns an address that does not have a DaikonPtrInfo already in the basemap, it initializes the `birthcount` to 1. However, if `DAIKON_malloc()` returns an address that already has an associated DaikonPtrInfo, it increments the `birthcount` in the new DaikonPtrInfo. This is necessary to keep dangling pointers (pointers that point to a block that has been `free()`d) from accidentally becoming valid, simply because the block they point to has been renewed by `DAIKON_malloc()`. This newly-allocated block is semantically a different block than the old one, and dangling DaikonSmartPointers that point to it should not be considered valid. When a DaikonSmartPointer is assigned or created, it looks up its associated DaikonPtrInfo in the basemap, and the `birthcount` field from the DaikonPtrInfo is copied to the DaikonSmartPointer object's `birthcount` field. Then, when a DRT output procedure tries to output a DaikonSmartPointer's contents, it checks to see if the `birthcount` in the DaikonSmartPointer matches the `birthcount` in the DaikonPtrInfo. If they match, it goes forward with output, but if they don't match, the pointer contents are output as uninit. This keeps stale pointers (which could even be a different type than the new block!) from incorrectly outputting

```
void *DAIKON_malloc(size_t size) {
  void * ret = malloc(size);
  if (ret) {
    if (daikonBasemap[ret] == NULL) {
      /* this pointer isn't in the basemap yet. */
      daikonBasemap[ret] = DaikonPtrInfo
(/*max_seen=*/ret, /* We haven't seen anything yet */
 /*refcount=*/1, /* note! fake refcount must get --'ed by free() */
 /*modbit=*/1, /* The memory is valid - */
 /*bounds=*/(void*)(((char*)ret)+size), /* - up to this point */
 /*leeway=*/0,
 /*from_malloc=*/true,
 /*unformatted=*/true,
 /*birthcount=*/1); /* first time we've seen this block */
    } else {
      /* this pointer was, at one point, in the basemap. */
      int oldrefcount = daikonBasemap[ret].refcount;
      int oldbirthcount = daikonBasemap[ret].birthcount;
      /* If oldrefcount>0, then there are stale pointers still pointing
 to this block.  We need to add the old refcount into the new one,
 so when they get reassigned or go out of scope, their refcount
 decrementing won't break the new DaikonPtrInfo. */
      daikonBasemap[ret] = DaikonPtrInfo
(/*max_seen=*/ret, /* 'new' block - none of it is valid yet */
 /*refcount=*/1 + oldrefcount, /* refcount folding */
 /*modbit=*/1,
 /*bounds=*/(void*)(((char*)ret)+size),
 /*leeway=*/0,
 /*from_malloc=*/true,
 /*unformatted=*/true,
 /*birthcount=*/1 + oldbirthcount);
    }
  }
  return ret;
}
```

Figure 3-4: Psuedocode for the `malloc()` wrapper function.

`DAIKON_malloc()`ed blocks.

3.5.3 Role of DaikonSmartPointer

The smart pointer class has a constructor that takes an argument of type `void *`, the return type of `malloc()`. Psuedocode for this constructor is shown in Figure 3-5. When this constructor is called, it first looks up the block in the basemap to see if it's already there, and if so, whether the `from_malloc` flag is set. If it is, it knows that it has to "format" the memory region.

Since smart pointers are templatized, the constructor for the smart pointer knows the type of struct (or scalar) that's being pointed to. The type of the block is not known at the time `DAIKON_malloc()` is executed, so all type-aware operations must be handled by DaikonSmartPointer. The DaikonSmartPointer constructor does the three type-aware operations: It constructs elements of its pointed-to type in the memory block, and sets up a type-templatized destructor and copy function pointer for later, by setting the `dest` and `copy` fields in the basemap entry. The first thing the constructor does is iterate over the memory block, using placement `new` to construct each struct in-place. Placement `new` calls the default constructor for the struct, which constructs each smart pointer field as well, maintaining coherence. By dividing the size of the memory region by the size of the type, it knows how many elements fit in the memory region, and constructs exactly that many.

After the block of memory has been formatted, the basemap entry is marked accordingly, so `DAIKON_free()` knows that a destructor must be called. The `dest` field of the DaikonPtrInfo is a function pointer, which is set to point to a template function that will iterate over the same block, destroying the constructed fields. The `copy` field is also a function pointer, which is set to point to another template function that copies elements of the template type to another memory region, explicitly constructing the objects as they arrive in the destination region. See Figure 3-7 for what these functions look like.

3.5.4 Role of `DAIKON_free()`

When `DAIKON_free()` is called on a pointer, it calls a method of DaikonSmartPointer called `free_self()`. This is to ensure that that the correct block is passed to the `free()` system call, even if the `index` field of the smart pointer points further into the block. This is a necessary consequence of array padding, which will be explained in Section 3.8.2.

The `free_self()` (see Figure 3-6) method performs a few consistency checks with the associated basemap entry. First, it checks `from_malloc` to make sure that the block was originally allocated with `DAIKON_malloc()`. If `from_malloc` isn't set, the

```
DaikonSmartPointer<T>::DaikonSmartPointer(const void* toCopy) {
  /* find the first entry greater than the key. */
  DaikonBasemap::iterator dbi =
    daikonBasemap->upper_bound(toCopy);
  --dbi;  /* scan backwards to find the last entry <= the key */
  if ((dbi==daikonBasemap->end()) ||
      (dbi->second.bounds < (void*)toCopy)) {
       /* New entry.  Base should equal index, and we should
          set it up in the basemap. */
    base = toCopy; index = toCopy;
    dbi->second = DaikonPtrInfo
      (/*base=*/toCopy, /*refcount=*/1, /*modbit=*/1);
  } else {
    /* toCopy falls within the bounds of an existing block.
       Let's construct off that block.  Take the basemap key
       as our new base, and set the index to be what we*/
    base = (const T*)dbi->first;
    index = toCopy;
  }
  DaikonPtrInfo & myinfo = dbi->second;
  birthcount = myinfo.birthcount; /* copy birthcount from the basemap */
  if (myinfo.from_malloc && myinfo.unformatted_block) {
    /* check basemap, construct T's in-place if necessary.  We -don't-
       construct in the leeway zone, that's done by DSA. */
    for (T *iter = (T*)toCopy; iter<myinfo.bounds; iter++) {
      new(iter) T();
    }
    /* link back to the array destructor and copier */
    myinfo.dest = &(daikon_kill_type<T>);
    myinfo.copy = &(daikon_copy_type<T>);
    /* we've formatted it now. */
    myinfo.unformatted_block=false;
  }
  myinfo.modbit=1;
};
```

Figure 3-5: Psuedocode for the smart pointer constructor that performs in-place construction on a block returned from `DAIKON_malloc()`.

program aborts. It then checks to make sure that the `modbit` isn't 2, because a pointer cannot be `free()`d multiple times. If both the consistency checks pass, it checks the `unformatted` variable. It's possible that `unformatted` is still true (for instance, if the memory block was passed to and used only by functions outside of instrumentation scope, and therefore never constructed by any smart pointer assignment), in which case no destructor is called. If `unformatted` is false, and the `dest` field is non-null, the destructor is called. The destructor (see Figure 3-7) is a wrapper function that iterates over the block, individually calling the destructors of each constructed element. If

```
template <class T>
void DAIKON_free(DaikonSmartPointer<T> ptr) {
/* All this is farmed out to smartpointer.free_self(), since
   we need to know the base to do this. */
  ptr.free_self();
}

template <class T>
void DaikonSmartPointer<T>::free_self() {
  DaikonPtrInfo &myinfo = basemapLookup();
  if (myinfo.unformatted_block) {
    /* it's still unformatted - either it's full of scalars, or structs
       never got constructed in it.  either way, you can just free(). */
  } else if (myinfo.dest) {
    /* it's been formatted somehow, and there's a destructor for it.
       call the destructor. */
    (*(myinfo.dest))((void*)base, myinfo.bounds);
  }
  (myinfo.refcount)--;
  myinfo.modbit = 2;
  free((void*)base);
}
```

Figure 3-6: Psuedocode for the `free()` wrapper function.

the elements are smart pointers, or structs that contain smart pointers, then their destructors are called, and the refcounts of their pointed-to memory blocks will be decremented, maintaining consistency.

Finally, the memory is deallocated. The refcount of the block is then decremented to account for the increment by the `DAIKON_malloc()` call earlier. The refcount is not set to zero, because even though the block is invalid, there is at least one smart pointer (the argument to `DAIKON_free()`) that is still pointing to the block, and the destructors for those smart pointers will eventually be called and the refcount decremented to zero on its own. The `birthcount` is left alone, and is not incremented until the next time this particular block is returned by `DAIKON_malloc()`. The modbit field, however, is set to 2, which is means that the data is invalid. Any output handler that reaches this block during output to the `dtrace` file will see the modbit, and record to the `dtrace` file that the data is uninitialized.

```
/* basically, array delete for in-place constructed objects,
   templatized on class. */
template <class T>
void daikon_kill_type(const void *victim, const void *top) {
  for (T *iter = (T*)victim; iter<top; iter++) {
    iter->~T();
  }
}

/* this is called by DAIKON_memcpy(), DAIKON_bcopy(),
   DAIKON_memmove(), etc. */
template <class T>
void *daikon_copy_type(DaikonSmartPointer<T> destarg,
       DaikonSmartPointer<T> srcarg,
       size_t n) {
  /* Safe for overlapping areas of memory - copies in the right
     direction, and through an intermediary. */
  T intermediary;
  DaikonSmartPointer<T> dest=destarg, src=srcarg;
  int numelements = n / sizeof(T);
  if (dest<src) {
    /* copy ascending. */
    for (int iter=0 ; iter < numelements; iter++)
      dest[iter] = intermediary = src[iter];
  } else {
    /* copy descending. */
    for (int iter=numelements-1 ; iter >= 0; iter--)
      dest[iter] = intermediary = src[iter];
  }
  return dest;
}
```

Figure 3-7: Destructor and copier functions for `DAIKON_malloc()`'ed memory regions.

3.6 DRT initialization

The instrumented code relies heavily on the DRT, so it is essential that the DRT is initialized before the user code begins execution. The DRT has two global variables that are linked into the instrumented program. The first is the basemap itself, which is needed by every smart pointer. The second is the `FILE *` representing the `dtrace` file, which is necessary for output at every PPT.

The `dtrace` file needs to be opened before the first output call is encountered. The basemap needs to be constructed even earlier, in case there's a pointer variable in the global scope, which would be instrumented as a smart pointer. For safety, both of these undergo construction before execution of the `main()` function even begins.

To ensure that these objects are initialized properly, I used a design pattern from Meyers [Mey92] for ensuring early construction of global objects.

At the end of `daikon_runtime.h`, which is automatically preincluded in every instrumented file, there is a definition of an object of type DaikonInitializer. This object gets constructed before any smart pointer is (because it appears before any global in program source order), and before the entry to the `main()` function. This object opens the `dtrace` file and constructs the basemap, and when the program terminates and it goes out of scope, it calls the destructors as well.

3.7 Struct instrumentation

All user-defined structs and classes are also output to the `dtrace` file, but they have special output handlers that are generated by the `dfec` executable at instrumentation time (see Figure 3-8). A "maximum instrumentation depth" is specified by the user, and `dfec` does a breadth-first traversal of the IL tree branch corresponding to that type, up to the user-specified depth. Entries for all fields touched by this traversal are output to the `decls` file.

`dfec` also generates, for each struct type, two functions that write output to the `dtrace` file for an instance of that type. The first is a static, global-namespace function called `daikon_output_user_type()`, which `dfec` generates calls to. It checks to make sure that the struct instance provided is initialized (i.e. has modbit 1). If the argument is uninitialized, it outputs dummy values to the dtrace file for each field in the struct. If the argument is initialized, `daikon_output_user_type()` then proceeds to call the second function, `daikon_output_thyself()`. `daikon_output_thyself()` is a member function of the struct type, and when called, outputs the actual contents of its instance's fields. Struct type output is divided into two functions for two reasons. The first is that a global namespace function like `daikon_output_user_type()` may not have access to all of a struct's fields, if some are private, so a member function is necessary. The second is that, if the instance is uninitialized, calling a member function of it could have adverse effects. This is why both functions are necessary.

Each instance of `daikon_output_user_type()` takes five arguments: a `name`; a pointer to a structure `var`, an integer `depth_left`, and two booleans, `from_pointer` and `is_array`. `var` is the struct that is to be output to the `dtrace` file. The `name` argument is simply the name of the structure, and is prepended to each field being output (for instance, for a struct pointer named `head` and a field named `data`, the `name` argument would be "head" and the function would output "head-¿data" to the `dtrace` file).

The `from_pointer` boolean variable tells the function whether the struct being output was originally a pointer, or if its address had to be taken when calling the function (in the example above, had `from_pointer` been false, it would have output "head.data" instead of "head-¿data" to the `dtrace` file). `is_array` is only true when `from_pointer` is true, and indicates that the `var` argument is an array of structs, and that the function should output arrays of the type's fields instead of treating each as a single occurrence.

The `depth_left` and `modbit` arguments come into play when structs have fields that are also structs. Since `daikon_output_user_type()` outputs each field of the `var` argument, it must generate a call to `daikon_output_user_type()` if one of the fields is another struct. This could be a call to another instance of `daikon_output_user_type()` if the field is of a different struct type than `var`, or it could be a recursive call, as in Figure 3-8, if the field is of the same struct type. Recursive calls will show up in circular data structures like linked lists. `depth_left` is decremented with each call to `daikon_output_user_type()`, so that struct field traversal eventually bottoms out at the same user-defined depth as the IL tree traversal did when the `decls` file was generated. An example of a generated `daikon_output_user_type()` function is shown in Figure 3-8.

If a null or invalid pointer is encountered during the traversal, the DRT must still continue outputting entries to the `dtrace` file, as Daikon requires that there be a matching `dtrace` entry for every entry appearing in the `decls` file. The `daikon_output_user_type()` function calls a member function of DaikonSmartPointer called `invalid()` to determine if it's null, or points to an uninitialized block. If it is, `daikon_output_user_type()` outputs dummies for its fields, and effectively outputs dummies for its children's fields by calling `daikon_output_user_type()` with the `var` argument set to null. If the pointer is valid, however, it calls `daikon_output_thyself()`, which does the actual work of outputting the fields in much the same manner.

3.8 Safeguards

During the testing of `dfec`, it became apparent that many C programs were unsafe, and relied on the benevolence of the compiler and their runtime environment to work correctly. `dfec` was designed to be robust when faced with certain common programmer errors, and to alert the programmer of the bug so it can be fixed.

3.8.1 Fixed-length array overrun protection

A few programs in the Siemens suite [RH98], which we used for testing, contain fixed-length array overrun errors [HME02]. This is one of the most common errors in C/C++ programming, and can be hard to track down. The effect of an access beyond the declared bounds of an array is undefined, but in practice, it doesn't always have an adverse effect. The `schedule` program, for example, contains an array that the `gcc` linker places at the end of the program's memory space. A write beyond the bounds of the array, as long as it isn't too far, just writes into unused slots of the memory page allocated for the program. This doesn't affect any other program data, in this case.

However, when instrumented, the array is not the last thing on the page. The `gcc` linker places the data structures for the DRT immediately following the `schedule` data structures, so when the array is overrun, DRT data is overwritten, corrupting it and frequently causing the program to crash. Two safeguards were implemented to deal with this situation.

Firstly, fixed-length arrays in user code are changed to a parameterized type called DaikonSmartArray when instrumented. DaikonSmartArray is a derived class of DaikonSmartPointer, and does all the normal smart pointer operations such as refcounting and keeping track of extent for invariant detection purposes. In addition, it allocates the array at construction, keeps track of its declared length, and stores the

```
|    Original Source            |    psuedo-decls file                   |
+===============================+========================================+
| struct list {                 | std.foo(list *;)int:::ENTER            |
|   int data;                   | pointer (hashcode)                     |
|   struct list *next;          | pointer->data (int)                    |
| };                            | pointer->next (hashcode)               |
|                               | pointer->next->data (int)              |
| struct list arr[10];          | pointer->next->next (hashcode)         |
|                               | pointer->next->next->data (int)        |
| int foo(struct list *ptr) {   | pointer->next->next->next (hashcode)   |
|   ...                         | ::arr (hashcode)                       |
|                               | ::arr->data[] (int array)              |
|                               | ::arr->next[] (hashcode array)         |
+===============================+========================================+

static void daikon_output_user_type(string name,
                                    DaikonSmartPointer<list> var,
                                    int depth_left, int from_pointer,
                                    int is_array) {
  string separator =
    (from_pointer ? "->" : ".");
  if ((depth_left)>0) {
    if (var.invalid(/*test_contents=*/true)) {
      if (is_array) {
        daikon_output_dummy(name + separator + "data[]");
        daikon_output_dummy(name + separator + "next[]");
      } else { /* !is_array */
        daikon_output_dummy(name + separator + "data");
        daikon_output_dummy(name + separator + "next");
        daikon_output_user_type(name + separator + "next",
                                (DaikonSmartPointer<list>)0,
                                 depth_left-1, /*from_pointer=*/1,
                                 /*is_array=*/0);
      } /* is_array */
    } else { /* !var.invalid */
      list::daikon_output_thyself(name, var, depth_left, separator,
                                  is_array);
    } /*if(invalid)*/
  } /*if(depth_left)*/
};/*daikon_output_user_type*/

void list::daikon_output_thyself(string name,
                                 DaikonSmartPointer<list> var,
                                 int depth_left, const char *separator,
                                 int is_array) {
  if (is_array) {
    daikon_output_smartpointer_ints
      (name + separator + "data[]",
       DaikonSmartPointer< int >(&(var->data)),
       /*spacing=*/sizeof(list));
    daikon_output_smartpointer_pointers
      (name + separator + "next[]",
       DaikonSmartPointer<DaikonSmartPointer<int> >(&(var->next)),
       /*spacing=*/sizeof(list));
  } else { /* !is_array */
    daikon_output_int(name + separator + "data", int(var->data));
    daikon_output_pointer(name + separator + "next", var->next);
    daikon_output_user_type
      (name + separator + "next", (DaikonSmartPointer <list>)var->next,
       depth_left-1, /*from_pointer=*/1, /*is_array=*/0);
  } /* is_array */
};
```

Figure 3-8: An example of a dfec-generated user type output handler.

length in the `bounds` field of the DaikonPtrInfo structure (see Figure 3-3) associated with the address of the beginning of the array. That way, no matter what pointer is derived from the array, the fixed length information is maintained. For smart pointers (and smart arrays, by extension), the pointer dereference and array subscript overloaded operators check the basemap for this `bounds` field, and if the subscript (or offset) being accessed is beyond the acceptable bounds, the instrumented program will `abort()`. This doesn't save the program, but it alerts the programmer to a bug in the code that needs to be fixed before invariant detection can proceed. This detects the bug at the point of error instead of manifesting later, such as an inexplicable crash due to DRT data corruption, which would be hard (if even possible) to debug.

The second method of protection against fixed-length array overrun is a guard structure that surrounds the DRT variables in memory. Specifically, the `FILE *` representing the `dtrace` file (the variable that the linker puts at the low end of memory) has an integer variable declared before and after it. At program initialization, the integers are initialized to two "magic numbers," and before every access to the `FILE *`, these two numbers are checked to make sure that they haven't changed. If the user program accesses data sequentially, it will corrupt these numbers before breaking the object. In this event, the instrumented program will also `abort()` with a message that informs the user that DRT data has been corrupted. This safeguard is useful for when a completely random pointer access destroys DRT data, or for when a fixed-length buffer is passed to a function outside of instrumentation scope (such as `gets()`).

3.8.2 Array padding

Array padding is an extension of the fixed-length array overrun protection described in Section 3.8.1. The user can opt to add a small, user-selectable number of elements to the beginning and end of every fixed-length array, by defining a preprocessor macro named `DAIKON_FIXED_LENGTH_ARRAY_PADDING` to the number of extra array elements desired. Then, if the instrumented code writes beyond the bounds of the array in either direction, the runtime library can simply warn the user instead of aborting, if the access falls within these padding zones. Since most array accesses are sequential, the user often can get away with a few overruns, and by defining the size of the padding zone (or "leeway") correctly, the user can make even an incorrect program exhibit well-defined behavior.

Array padding is implemented in two parts of the runtime library. The first part is the allocation, which is taken care of in the constructor for DaikonSmartArray. When the array is allocated, an extra

$$2 * element\ size * \texttt{DAIKON\_FIXED\_LENGTH\_ARRAY\_PADDING} \tag{3.1}$$

bytes are added to the allocation. Since DaikonSmartArray is derived from DaikonSmartPointer, it has both a `base` and an `index` field. The `base` is set to point to the bottom of the allocated block, and the `index` field is set to point *element size* * `DAIKON_FIXED_LENGTH_ARRAY_PADDING` bytes into the newly-allocated block. Since the `index` field is is used to represent the "actual" pointer, there are `DAIKON_FIXED_LENGTH_ARRAY_PADDING` valid elements before the user-visible beginning of the array, as well as `DAIKON_FIXED_LENGTH_ARRAY_PADDING` valid elements beyond the user-visible end of the array.

The second part of array padding is the warnings that are issued when the instrumented code accesses an element in the padding area of the array. Since all dereferences are handled by DaikonSmartPointer, the checking is done in a DaikonSmartPointer member function called `note_and_dereference`. `note_and_dereference` looks up the relevant block in the basemap, and checks its DaikonPtrInfo for the `bounds` field. If `bounds` is set, it first checks to see if the access falls outside the bounds, and if so, throws a fatal error. If the access is within the bounds, it then checks the `leeway` field, to see if the access falls within the padding zone. If it does, the runtime library issues an error, but proceeds with the access. `max_seen` is updated, and a reference to the appropriate memory address is returned. Code for `note_and_dereference` is shown in Figure 3-9.

```
T & DaikonSmartPointer<T>::note_and_dereference(int ofs) {
  DaikonPtrInfo &myinfo = basemapLookup();
  if (invalid(/*test_contents=*/false)) {
    fprintf(stderr,
     "daikon_runtime: program attempted to access an invalidated block!\n");
    exit(DAIKON_ERR_CODE);
  }
  if (myinfo.bounds) {
    /* The leeway calculations are as follows:
       The range [base, bounds) is the only possible addressable range.
       (leeway) bytes inwards from that is allowed, but reported. */
    if (((index+ofs)>=myinfo.bounds) ||
        ((index+ofs)<base)) {
      report_array_overflow(ofs, /*fatal=*/true);
      exit(DAIKON_ERR_CODE);
#if DAIKON_FIXED_LENGTH_ARRAY_PADDING != 0
    } else if (((int)(index+ofs)>=((int)(myinfo.bounds))-myinfo.leeway)||
               ((int)(index+ofs)<((int)base)+myinfo.leeway)) {
      /* it's not beyond the bounds as tested above, but it's
         in the leeway zone. */
      report_array_overflow(ofs, /*fatal=*/false);
#endif /* DAIKON_FIXED_LENGTH_ARRAY_PADDING != 0 */
    }
  }
  if (index+ofs >= myinfo.max_seen)
    myinfo.max_seen = index+ofs+1;
  return (T&)(index[ofs]);
};
```

Figure 3-9: The bounds-checking routine for DaikonSmartPointer.

3.8.3 Automatic scalar initialization

The space program from the Siemens suite [RH98] makes the assumption that scalar values are initialized to zero when declared. This behavior is not part of the ANSI C standard, though some compilers (notably gcc) do this anyway. However, the version of g++ that dfec is compatible with does not. When instrumenting a C program that assumes gcc is going to initialize its variables to zero, the compilation of the instrumented source produces a version of the program where undefined behavior manifests itself as a bug.

Though this is an error on the user's part, the user will rationally attribute any change in behavior in the instrumented program to a fault in dfec. Currently, dfec outputs an zero initializer for every scalar variable that the user has left uninitialized. This allows the space program, as well as other programs that make the same assumption, to work correctly.

3.9 Other Features

3.9.1 Disambiguation

The C language allows a programmer to use a pointer variable as either a simple reference to one variable, or as a base pointer to a sequential array of variables of the same type, as demonstrated in Figure 3-10. The runtime library can generally determine what memory is valid, and usually pointers to single variables are simply output as arrays of length 1. However, this is not always good enough. If the user has a pointer to a single `int` variable, and uses the pointer to address an `int` field inside a larger struct with other fields (which may not necessarily be `int`s), the runtime will incorrectly output the subsequent bytes in the struct as if they were members of an `int` array. Also, there must be a decision made at instrumentation time whether a struct pointer is an array or a pointer to a single struct. If it is instrumented a pointer to a single struct, then fields of other structs in the array will not be output. However, if it is a single struct but instrumented as an array, recursive instrumentation will be shallower than in the single case - array fields of the struct cannot be output, since the struct is already an array, and Daikon proper cannot handle two-dimensional arrays. Clearly, a mechanism to allow the user to select whether to instrument a pointer as a pointer-to-single or as an array is necessary.

While static analysis could guess as to what the programmer intended, this is beyond the scope of this thesis, and is not implemented. However, `dfec` provides a mechanism by which the user can specify, for a specific variable at a specific program point, how that variable should be output. This feature is called "disambiguation." Disambiguation is controlled by a separate file that is loaded at instrumentation time, containing a list that specifies the representation types of dereferenced pointer variables. This way, a static analysis utility could be written later to use this mechanism, but remain over-ridable by the programmer.

In Figure 3-10, `many_characters` refers to an array while `single_character` refers

```
...
char single_character;
char many_characters[100];

char *pointer;
pointer = &single_character;
pointer = many_characters;
...
```

Figure 3-10: The versatility of pointer variable types.

to a single value. Information about whether each pointer refers to an array or a single element can be specified in a "disambig file" that resides in the same directory as the `decls` file. `dfec` has a command-line option that causes it to read this file instead of assuming all pointer variables should be instrumented as their default types. (`dfec` can also produce the file automatically, permitting users to edit it for use on subsequent runs, rather than having to create it from scratch.)

The disambig file lists all the instrumented PPTs, and under each, a list of all the variables in scope at that PPT, along with the types that the variables are instrumented as. For pointer variables, there are two options: "A" for array, and "P" for pointer to single value. For variables of type `char`, there are two options: "I" for integer (i.e. numerical value), "C" for character. This allows the user to choose whether a `char` variable refers to a short numerical value that mathematical invariants should be calculated over, or an ASCII character. An example is shown in Figure 3-11, where the desired type of `array` is "A" (an array of values), and the desired type of `result` is "P" (a pointer to a single value).

The `disambig` file contains a single entry for the global variables, another entry for the parameter variables at each PPT, and another entry for every user-defined class or struct in instrumentation scope. Members of structs or classes can be disambiguated just like variables at a program point. Entries can occur in the `disambig` file in any order, and variables can be omitted, which will lead to them being instrumented as their default type.

```
/* example program: */                 | /* ideal disambig file: */
                                       | std.peq(int *;int *;)int:::ENTER
int foo[10] =                          | p1
  {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}; | P
                                       | p2
int peq(int *p1, int *p2) {            | P
  return (*p1 == *p2);                 |
}                                      | std.peq(int *;int *;)int:::EXIT
                                       | p1
int main() {                           | P
  int i;                               | p2
  for (i=0; i<9; i++)                  | P
    peq(foo+i, foo+i+1);               |
}                                      | ...


//Invariants discovered, using disambiguation
===========================================================================
std.peq(int *;int *;):::ENTER
*p1 <= *p2
===========================================================================
std.peq(int *;int *;):::EXIT1
p1 == orig(p1)
*p1 == orig(*p1)
p2 == orig(p2)
*p2 == orig(*p2)
return one of { 0, 1 }
*p1 <= *p2

//Invariants discovered, without using disambiguation
===========================================================================
std.peq(int *;int *;):::ENTER
size(p1[])-1 == size(p2[])
p1[] elements >= 1
p1[] sorted by <=
p2[] sorted by <
p1[] < p2[] (lexically)
p2[] is a subsequence of p1[]
p1[] >= ::foo[] (lexically)
p2[] > ::foo[] (lexically)
===========================================================================
std.peq(int *;int *;):::EXIT1
p1 == orig(p1)
p1[] == orig(p1[])
p2 == orig(p2)
p2[] == orig(p2[])
size(p1[])-1 == size(p2[])
p1[] elements >= 1
p1[] sorted by <=
p2[] sorted by <
return one of { 0, 1 }
p1[] < p2[] (lexically)
p2[] is a subsequence of p1[]
p1[] >= ::foo[] (lexically)
p2[] > ::foo[] (lexically)
```


Figure 3-11: An example program, its `disambig` file, and invariants with and without using disambiguation.

3.9.2 Variable Comparability

Variable comparability information can speed up the invariant detection process and reduce the number of trivial invariants by eliminating comparisons between unrelated variables. For instance, if two variables have the same type, but are never actually compared (directly or indirectly) in the target program, it can be assumed that any invariants discovered involving both variables are purely coincidental (see Figure 3-12). `dfec` can, after instrumentation, call an external program called Lackwit [OJ97], which performs static analysis on C code to determine what variables are comparable. A combination of Perl scripts and another preparatory front end that aids Lackwit in its analysis of array variables are used, and the comparability types are inserted into the `decls` file in the correct places. This process is described in Section 3.9.2. No variable comparability detection is implemented or planned for C++.

Figure 3-12 is a demonstration of the need for comparability types. Without comparability types, Daikon would attempt to calculate invariants relating `index` and `weight`, because they're both `ints`. However, since one is an index and one is a numerical value, their comparability types will be different, and no invariants will be calculated relating their values. However, `weight` will be compared to the elements of the `bar` array.

Lackwit execution

Lackwit, as distributed, contains a few bugs and is missing a few necessary features for it to be transparently usable as a variable comparability detector. Michael Harder has written an `lwpp` Perl script that makes the use of Lackwit transparent for the user, and I've written a separate extension of the EDG front end called `lh` that inserts information into a source file to allow Lackwit to better detect variable comparability.

Lackwit comes with a file called `libc.c`, which contains signatures for functions in libc, and fake stub implementations for them that causes Lackwit to associate their arguments in the correct manner. Every execution of `dfec` causes this file to be processed by Lackwit, which notes every time a variable is comparable to another (by an assignment, boolean relation, or other expression that involves the

```
int bar[] = {1,1,2,3,5,8};

int foo(int index, int weight) {
  return bar[index] + weight;
}


Invariants detected, using Lackwit
===========================================================================
std.foo(int;int;):::EXIT1
index == orig(index)
weight == orig(weight)
::bar == orig(::bar)
index >= 0
weight >= 0
::bar has only one value
::bar[] elements one of { 1, 2, 3, 5, 8 }
weight < return

Invariants detected, without using Lackwit
===========================================================================
std.foo(int;int;):::EXIT1
index == orig(index)
weight == orig(weight)
::bar == orig(::bar)
index >= 0
weight >= 0
::bar has only one value
::bar[] elements one of { 1, 2, 3, 5, 8 }
::bar[index..] >= (index)
::bar[index..] sorted by <=
::bar[0..index-1] elements one of { 1, 2, 3, 5 }
::bar[weight..] >= (index)
::bar[weight..] sorted by <=
::bar[weight+1..] > (index)
::bar[weight+1..] sorted by <
::bar[0..weight-1] elements one of { 1, 2, 3, 5 }
index <= return
index <= size(::bar[])-1
weight < return
weight <= size(::bar[])-1
return >= ::bar[index]
```

Figure 3-12: A demonstration of the necessity of comparability types.

```
/* original version */
void foo(int *array,
 int index,
 int value) {
  array[index] = value;
}

/* version processed by lh */
void foo(int array_index, int array_element, int *array,
 int index,
 int value) {
((array_index=index,
  array_element=array[array_index],
  array[array_index])) = value;
}
```

Figure 3-13: The changes made by `lh` to a sample function.

two variables directly), and adds this information to a database that can later be queried to determine equivalence classes of variable comparability.

For each source file being instrumented by `dfec`, a copy is set aside and processed by other utilities before being input to Lackwit. First it is processed by `lh`, which takes in C source and outputs a modified version of the source. Lackwit alone does not calculate the correct comparability of array indices over function-call boundaries, so `lh` manually adds arguments to function calls for these indices, as demonstrated in Figure 3-13. Every array subscript operation is rewritten in a manner that the index is associated with a new local variable that `lh` adds, called `arrayname_index` for an array variable named `arrayname`. A similar rewriting is done for array elements, using a new variable called `arrayname_element`. This adds information that allows Lackwit to connect comparability classes over function-call boundaries.

Once all the source files have been handed to Lackwit and the comparability database has been built, the `decls` file is rewritten by the `lwpp` script that iteratively queries the Lackwit database for comparability identifiers of each variable. Because of bugs in the Lackwit executable, the transitivity of variable comparability is not always maintained, and the script performs additional unification of comparability classes. Another bug causes the Lackwit executable to crash if a field of a structure is queried when it hasn't been explicitly referenced in the source. `lwpp` catches these crashes, and calculates comparability of struct fields by referencing the comparability classes of its parent.

Chapter 4

Experience

The two major requirements of the `dfec` system are that it does not affect the behavior of the instrumented program, and that it outputs as much quality data as possible to the `dtrace` file so that Daikon proper has can discover quality invariants.

A test suite has been developed that consists of six programs from the Siemens suite [RH98], along with the reference implementations of the MD5 cryptographic hash [Plu99] and Rijndael [RD01] algorithms. The Siemens programs are each accompanied with quite large test suites that fully exercise program behavior, and the Rijndael implementation is accompanied by an fairly exhaustive test stub. Correct program behavior is checked by comparing the output of the instrumented version with the uninstrumented one. Output behavior is harder to check. A regression test suite has been constructed that contains versions of `dtrace` and `decls` files that are known to exhibit quality invariants in the Siemens suite. The quality of these invariants is known, because they are expressions in the grammar of Daikon that have been proved, in other papers about the same suite. A `dtrace-diff` tool exists to compare two `dtrace` files and enumerate any differences between them, so it's easy to tell if a change in `dfec` yields a positive change in the `dtrace` file (such as suppression of the output of uninitialized variables) or a negative one (such as the failure to output structs). An invariant diff tool also exists, written by Michael Harder, which can show how the changes in the `dtrace` or `decls` files add or remove from the potential invariants that Daikon proper can detect. The invariant diff tool is used mostly to detect errors or changes in Daikon proper, and is outside the scope of this thesis.

4.1 Tools

4.1.1 `dtrace-diff`

For regression testing purposes, it is important to compare the output `dtrace` file produced by one run of an instrumented program to an earlier, 'ideal' version. The `dtrace-diff` tool is a Perl script written for this purpose. It is useful because it can catch bugs in the output procedures, or if a revision of the DRT is buggy and causes the instrumented program's behavior to differ from that of the uninstrumented version, it will catch that by noting a difference in expected PPTs. It is analogous to the UNIX `diff` utility, in that it compares two files and outputs lines where they differ. Unlike `diff`, however, it takes the semantics of the `dtrace` file into consideration. It takes an extra argument of a `decls` file, so it knows the types of the variables being compared. This is useful because some textual differences in the `dtrace` file may not indicate an actual semantic difference that should be worried about.

Floating-point variables can be compared to within a tolerance, so that they'll be considered to be equivalent if they're within a configurable ϵ of each other. This is useful for comparing the operations of an instrumented program on two different platforms, such as Linux and Solaris, where the floating-point hardware or library implementation may give slightly different results.

Pointer variables are recorded to the `dtrace` file as type "hashcode", analogous to Java where each object instance has its own unique hashcode that can be effectively implemented by just using its address in memory. These hashcodes are output to the `dtrace` file, but Daikon knows not to calculate mathematical invariants over them, since their values are non-deterministic and are tied to the memory layout of the system at the time of execution. Similarly, `dtrace-diff` knows that if two runs of a program have different values for a hashcode-type variable, it doesn't necessarily mean that there's a semantic difference in the two executions of the program. The exception to this is that null pointers are always the same value (namely, 0) on all executions of the program. `dtrace-diff` only outputs a difference on hashcode variables if one instance is null and the other is not.

4.2 Modifications to the test suites

The programs that were used as a test suite for `dfec` were occasionally uninstrumentable in their given form, due to size, faults, or dialect of C used. Below is a report of the modifications that were done to get them to work with `dfec`.

4.2.1 Siemens

The Siemens suite [RH98] is a suite of programs used frequently in the testing and program analysis field. They are written in the K&R dialect of C, and to instrument them, they first had to be passed through `protoize` [Gui]. Under instrumentation, some previously undetected faults were uncovered.

Array-overrun bugs were found in three of the Siemens programs (`print_tokens2`, `replace`, and `tcas`). These errors had not been noticed in previous research using the programs. When the coverage test suites were originally created, the erroneous programs had read or written an element beyond the bounds of an array without inducing a fault. However, in our environment, the array bounds errors caused the programs to crash, by corrupting the DRT's data structures and triggered an assertion failure. This was the motivation for adding the runtime checks for such memory errors to the DRT as described in Section 3.8.1. The implementation of the array padding feature (see Section 3.8.2) with the default setting of 2 padding elements allowed `print_tokens2` and `tcas` to execute without generating a fatal error. However, the `replace` program would require dozens of elements to operate correctly, so it still generates a fatal error. It is useful to have at least one program generating a fatal error in the test suite, solely to ensure that the runtime library's error-handling mechanism is operating correctly.

In addition to requiring protoization, the type system of some of the Siemens programs was slightly convoluted, and required some changes. For example, in `print_tokens2`, the two types `character_stream` and `token_stream` were occasionally used interchangeably. Both are typedef aliases of `FILE *`, and are compatible, but `print_tokens2` failed to include forward declarations for its functions. `dfec` failed to match the implicit definition of `unget_error(token_stream)` with the later explicit definition of `unget_error(character_stream)`. In ANSI C, this would have only generated a warning, because `token_stream` and `character_stream` are both pointer types. In the instrumented code, however, they are both smart pointers, which are template classes. The compiler has no way of knowing that they're compatible, and therefore gives an error. The function signature was changed to be consistently `unget_error(token_stream)` and it compiled and executed fine.

4.2.2 Rijndael

The Rijndael reference implementation [RD01] contains the algorithm and an extensive test stub. The test stub is designed for gathering timing data, so it runs an exceptionally large amount of code. In fact, it runs so much that the instrumented program (on Linux) eventually terminates abnormally, because the generated `dtrace` file exceeds the two gigabyte filesize limit on the ext2 filesystem. Because of this, we had to reduce the number of iterations of some top-level loops to get a test that would run in a reasonable amount of time, and produce a reasonably-size `dtrace` file for invariant detection.

Chapter 5

Future Work

`dfec` is a working system, and can handle non-buggy programs in a subset of the ANSI dialect of C. However, for usability purposes, this still leaves something to be desired. Non-buggy programs are extremely rare, and users may want to use `dfec` and Daikon specifically to find bugs in their programs. There is much room for usability work to be done, as well as increasing the subset of ANSI C that `dfec` can handle.

5.1 Additional `libc` instrumentation

Currently, only the `string.h` functions, `memcpy()`, `memmove()`, `bcopy()`, `getopt()`, `malloc()`, and `free()` have had wrappers written for them. There are other functions in `libc` that take pointer arguments, or return pointers, that should have wrapper functions written for them that understand the semantics of the function, and what effect it has on the heap.

5.2 `gcc` emulation

`gcc` has a few non-ANSI constructs in its header files, which occasionally require workarounds. Currently, `dfec` attempts to deal with these by defining system macros that make it act like an older version of `gcc` during the preprocessing step. This is not an elegant solution, and it may result in incorrect operation if function signatures in older areas of system header files don't match with new ones. Making `dfec` behave more like `gcc` would help streamline the instrumentation process. As EDG has just recently released a new version of their front end with better support for `gcc` emulation, moving `dfec` to be based on the newer version of EDG could help.

5.3 Unhandled C syntax

Because of the code that `dfec` inserts during instrumentation, sometimes the instrumented code is uncompilable. This is relatively rare, but there are a few cases where valid C is instrumented to become invalid C++. One example of this was given in Section 4.2.1, where two pointer types that should be compatible turn incompatible under instrumentation, because `gcc` does not follow the same type-compatibility rules for template objects as it does for pointer types.

5.4 Thread safety

The current implementation of the runtime library is not thread-safe, and consequentially, instrumentation of any threaded program will fail. To make the runtime library thread-safe, all accesses to the basemap will need to grab a mutex. This would avoid corrupting the STL map by calling mutation operations in parallel. Also, all DaikonSmartPointer operators assume that their associated DaikonPtrInfo will not change between the beginning and end of the function. A per-block mutex might work here, or possibly just reuse of the basemap mutex, so that only one DaikonSmartPointer could be executing an operator at a time.

Additionally, accesses to the dtrace file will need to be serialized, so that the output from two PPTs does not interleave. A simple mutex that must be grabbed before outputting a PPT, and is released afterward, should do the trick.

5.5 Same-sized smart pointers

Ideally, to minimize the possibility of error, and to maximize interoperability with uninstrumented code, it is desirable to make the smart pointer as much as possible like the primitive pointer it replaces. The current implementation of a smart pointer is an object which is a different size than a primitive pointer. This changes the memory layout of instrumented data structures. The instrumented code is adjusted accordingly, and should deal with the data structures transparently to the user, but uninstrumented code does not.

Currently, the `index` field of DaikonSmartPointer is placed at offset 0 in the memory layout of the object, so that uninstrumented code that thinks the object is a primitive pointer will overwrite this field. Since the `index` field represents the equivalent primitive pointer, this is more or less the desired behavior. However, when an array of pointers is instrumented, the size of the DaikonSmartPointer changes everything. Uninstrumented code iterating over the block of pointers will first get the `index` field of the first DaikonSmartPointer, then its `base` field, then its `birthcount` cast to a pointer, then the `index` of the second DaikonSmartPointer, and so on (See Figure 5-1). Clearly, this is incorrect behavior and will most likely cause the program to crash (when the uninstrumented code dereferences the `birthcount`, which is probably a low number, and therefore out of bounds).

A solution to this is to have the smart pointer object be the same size as the primitive pointer it replaces. The `base` and `birthcount` would be removed from the smart pointer, and put into another STL hashmap, separate from the basemap. Where the basemap maps from `base` pointers to attributes of a memory region, this new map would map uniquely from the address of a smart pointer variable in memory to a small structure holding its birthcount and its associated base pointer. On construction or assignment of a smart pointer, instead of copying the `base` and `birthcount` directly from the source smart pointer, the constructor function would make a new entry in the new map for the target smart pointer, and copy the `base` and `birthcount` fields from the source's entry in the new map. A diagram of this is shown in Figure 5-2.

This design would slow the instrumented executable down, but it would be much more safe. Additional safety features could be implemented, such as caching the last known value of the smart pointer in the new map, so if the pointer was changed by uninstrumented code, the runtime library could tell that it had been changed, and look to see if pointed to a different known block or was pointing to an unknown block

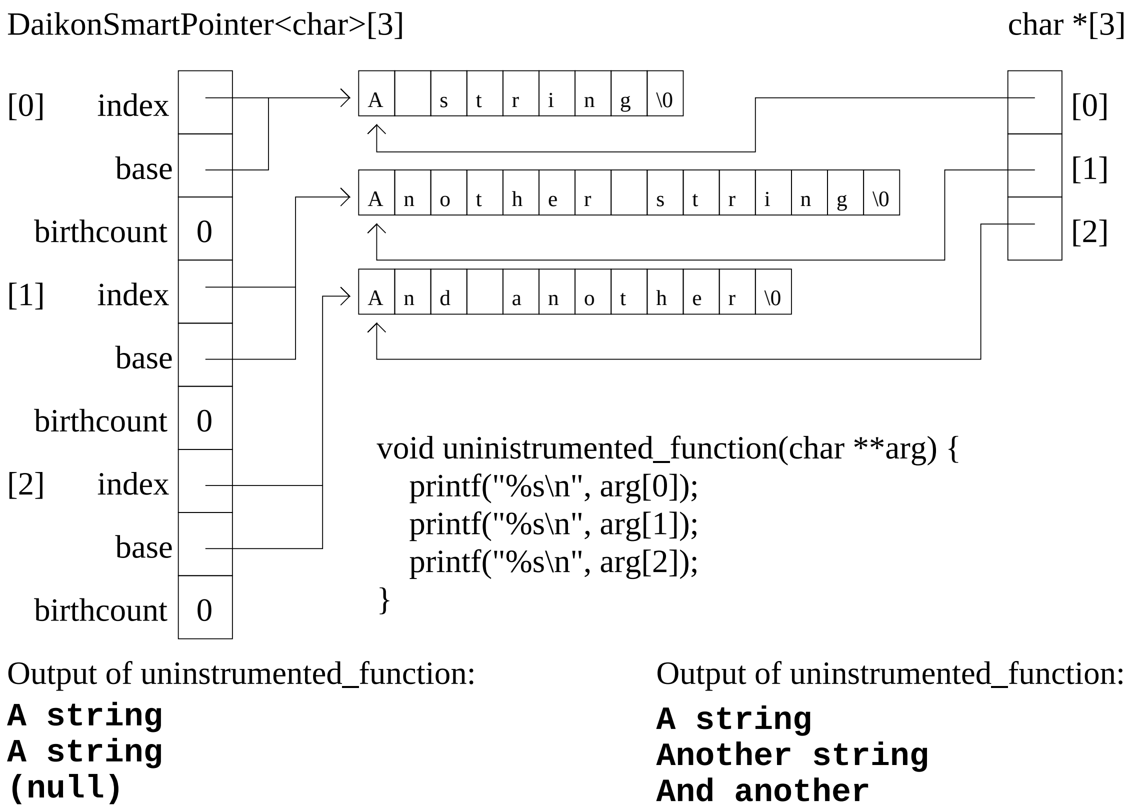

Figure 5-1: Size inconsistencies between primitive pointers and DaikonSmartPointer.

now.

5.6 Safety and debugging features

While `dfec` is not intended to be primarily used as a debugging tool, some features found in the tools discussed in Chapter 6 would be welcome additions to the runtime library. Saving a snapshot of the call stack and associating it with a `malloc()`ed block, like Valgrind [Sew], could help users track down a bug involving the block if it was found to impair program execution. Another Valgrind feature, which checks to make sure the program under analysis is using the deallocator (`free()`, `delete`, or `delete`[]) that correctly matches the allocator (`malloc()`, `new`, or `new`[]) used to obtain the given block. Finally, disclosing memory leaks at the end of the program, while it would neither aid invariant detection or help a faulty program execute, could possibly be of interest to users who are using the Daikon system primarily to fix bugs

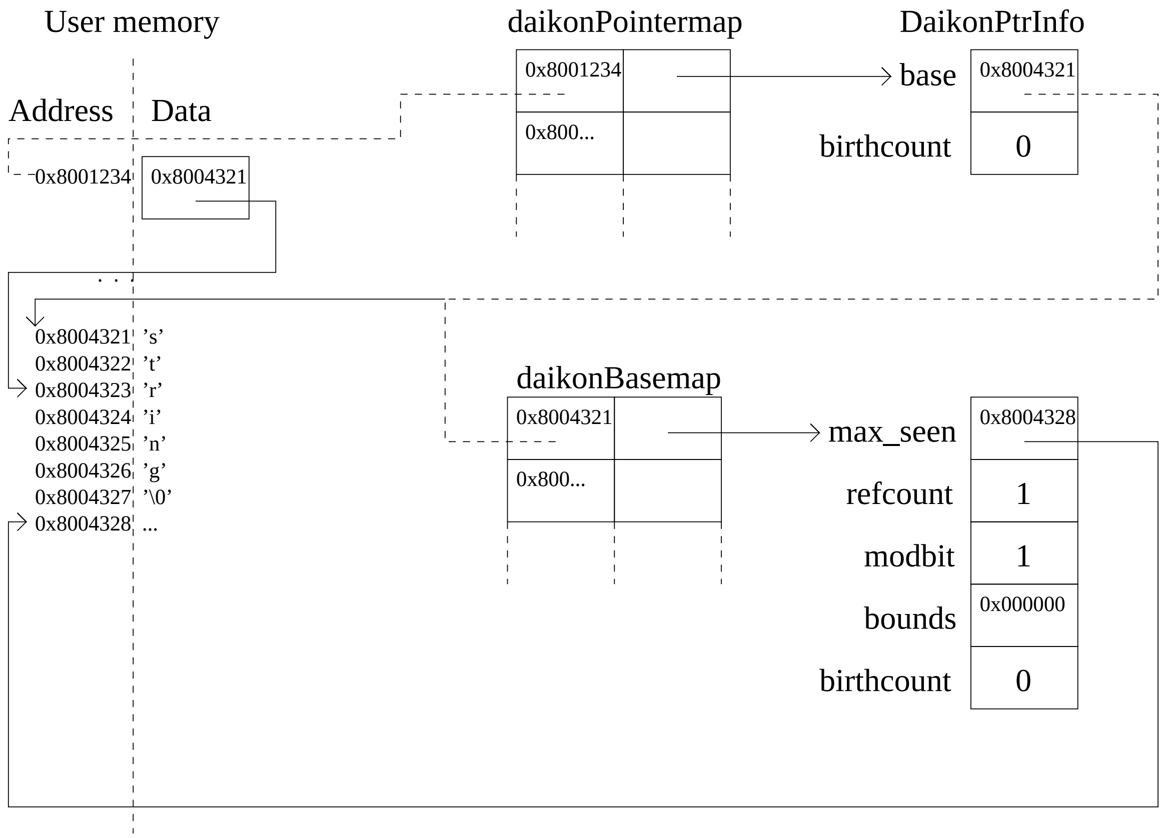

Figure 5-2: Proposed implementation of a slimmer smart pointer.

in their programs.

Chapter 6

Related work

`dfec`'s runtime library has much in common with testing tools such as Purify [HJ92], debug_malloc [Cah90], and Valgrind [Sew]. However, the aim of `dfec` is often opposed to the aim of these testing tools: while all monitor memory state in order to catch errors, `dfec` would like to allow the running program to execute as long as safely possible, even if it behaves illegally, while testing tools want to aggressively find and report errors and stop the program, even if the error would not normally be fatal.

6.1 Purify

Purify is a software testing and quality assurance tool that detects memory leaks and access errors [HJ92]. Its primary focus is to help the programmer discover and fix run-time detectable errors such as memory leaks (`malloc()`s without matching `free()`s), reads from uninitialized memory, and writes to `free()`d memory.

Purify is invoked on an executable file, modifying the object code in memory instead of modifying the source code. It inserts a function call before and after every data memory access instruction. At runtime, it keeps a state table that holds two status bits for every byte of memory in the heap, stack, and statically allocated data sections. One bit encodes whether a byte is writable or not (i.e. whether it's been allocated on the stack or `malloc()`ed and not yet `free()d`), and the other bit encodes whether the byte is readable or not (i.e. whether it has been initialized by a write). It also pads the beginning and end of `malloc()`ed blocks with a few bytes, but in contrast to `dfec`, it marks them as unreadable and unwritable, so if the program under analysis attempts to access them, it will report an error. `dfec`'s padding reports the error as well, but the intent is to allow the program to continue execution predictably and stably.

Also, Purify 'holds' memory blocks for a period of time after they've been `free()`d, in an attempt to minimize the chance that a block will be `free()`d, re-`malloc()`ed, and then accessed with a stale pointer. If this were to happen, this would be a memory error that Purify is unable to detect. `dfec` can detect this error, however, as the `birthcount` for the smart pointer and the `birthcount` for the memory block would not match.

Another error that Purify does not catch is when two array variables are adjacent in memory, and one overflows into the other. Purify attempts to minimize occurrence of this error by inserting unallocated bytes after variables as described above. Since most array accesses are sequential, this technique will most likely report an overflow before the program begins writing into the next array. However, if the program jumps far beyond the end of the array and writes into the next one without touching the unallocated region, Purify will not catch the error. `dfec` will catch it, because the base pointer for the array is associated with a DaikonPtrInfo that specifies the bounds. Any access beyond the bounds is caught.

In summary, `dfec` is able to catch certain errors that Purify is unable to due to their fundamentally different approaches in tracking memory state. Purify uses a byte-based approach, where each byte is associated with status bits that simply indicate the memory location's validity. The positive aspect of this is that the bookkeeping overhead of a memory access is a small constant (an array lookup, a test, and a write), whereas for `dfec`, the correct DaikonPtrInfo must be looked up, which is $O(\log n)$ in the current implementation. Additionally, it has finer granularity in determining validity. If a block is newly allocated, and then has a write performed on it several bytes in, Purify only reports the written byte as valid, where `dfec` will assume that all bytes from the beginning of the block up to the written byte are valid. Indeed, `dfec` performs construction and default initialization of array elements and elements of `malloc()`ed blocks as soon as possible, to avoid accidentally outputting garbage data. However, the memory overhead of Purify is a constant 25% of the user-addressible memory, whereas `dfec`'s memory usage can be much smaller, depending on the size of the allocated blocks. For a program that only has one pointer variable, that points to a `malloc()`ed block of four megabytes size, Purify uses an entire extra megabyte to store status bits, whereas `dfec` only uses a single DaikonPtrInfo structure, about 34 bytes (depending on architecture and compiler-generated padding for alignment purposes). Purify also catches errors that `dfec` does not, in specific, memory leaks. Memory leaks are of no interest to `dfec`. They could be easily caught by the runtime library by checking, during the destruction of the basemap, if there exist any DaikonPtrInfo structures for blocks marked as `from_malloc` that still have a `modbit` of 1, and complain. However, since a program will still run correctly and generate valid data even with a memory leak, `dfec` does nothing about it.

6.2 Valgrind

Valgrind [Sew] is a system that, like Purify, attempts to discover memory access errors. It operates in a similar fashion to Purify, and (like Purify) catches uses of uninitialized memory, accesses to memory that has been `free()`d, boundary violations on `malloc()`ed blocks, and memory leaks. Additionally, Valgrind catches reads and writes to inappropriate areas on the stack, and mismatched uses of `malloc()/new/new`[] versus `free()/delete/delete`[].

Valgrind, like Purify, has two bits per memory location - an A bit to indicate a valid address (i.e. if the address has been allocated), and a V bit to indicate valid data (i.e. if the address has been initialized). In contrast to Purify, Valgrind's validity-tracking operates at bit granularity, keeping status bits for every bit of user memory, and for every register in the processor. The A-bits (tracking address validity) are, as in Purify, per-byte. This leads to a program with a four-megabyte memory footprint (like the one described in Section 6.1) requiring an additional four-and-a-half-megabyte memory state table.

Valgrind's ability to detect incorrect accesses of the stack frame cannot be replicated by `dfec`. Mismatched uses of `malloc()/new/new`[] versus `free()/delete/delete`[]

could be detected by `dfec`, by adding status bits to DaikonPtrInfo that indicate how exactly the block was allocated. However, since `dfec`'s goal is to continue even in the face of programmer error, it does not perform these checks. Confusing `malloc()` for `new` and `free()` for `delete` is fatal, however. Array constructors and destructors are manually called by the runtime library on `malloc()`ed blocks, and these array constructors cannot be guaranteed to be compatible with `new`[] and `delete`[], which have compiler-specific implementations.

6.3 debug_malloc

debug_malloc [Cah90] is a library consisting of replacements for the standard system `malloc()`, `free()`, and related calls. No code change is required to use it, but the executable must be linked with the debug_malloc library. It works by providing hostile conditions for programs that will cause the program to segfault on certain memory errors when it might have continued unaware under normal conditions. The `malloc()` replacement fills the returned memory area with the value 0x01, which will break programs that expect memory areas to be initialized to zero, as well as programs which `free()` an area and expect it to be returned unchanged on the next `malloc()`. The `malloc()` replacement also allocates a certain amount of padding that it fills with "magic numbers", which it then uses when it performs consistency checks during `free()`, ensuring that these numbers have not been changed (which would indicate that the program wrote beyond the bounds of the block). The `free()` replacement also fills the block with the value 0x02, which will break programs that `free()` memory regions and then continue to reference them.

debug_malloc is extremely lightweight compared to `dfec`, Valgrind, and Purify. It requires very little additional state information, and therefore has a small footprint. Also, it doesn't check every access, instead only performing sanity checks during `malloc()` and `free()`. In exchange for the low overhead, some errors are missed. For example, it only catches out-of-bounds errors when the program actually writes out-of-bounds, whereas `dfec` can catch a read. The goal of debug_malloc is opposite to the goal of `dfec`, in that debug_malloc tries to make it hard for a buggy program to execute, and `dfec` tries to make it easier for even buggy programs to execute in the most "correct" fashion they can.

6.4 Run-Time Type Checking

The run-time type checking system devised by Loginov, Yong, Horwitz, and Reps [LYHR01] is more similar to `dfec` than the other testing tools described in that it performs instrumentation on the source code. Like the other testing tools, however, it maintains status bits for each byte of memory, instead of on a per-block basis, like with `dfec`. The status bits are used to represent what type the byte contains (one of "unallocated," "uninitialized," "integral," "real," and "pointer"). The "unallocated" and "uninitialized" tags work exactly the same way as they do in Purify, allowing the system to detect the set of memory-access errors that Purify does. The system's primary focus, however, is to detect and report run-time type conflicts. An example of a type conflict is when a union variable is written as an integral type, then is read as a pointer type. Statically, there is no feasible way to check that code is free from this type of error, but at runtime, the system notes that the union has been initialized to an integral type, and then issues a warning when it is read as a conflicting type.

`dfec` performs a nominal amount of runtime type-tracking, using the `dest` and `copy` function pointers in DaikonPtrInfo. Of course, this type-tracking is on a per-block basis, whereas the run-time type checking system is per-byte. Per-byte tracking allows for more complicated memory layouts (like non-homogeneous arrays for user-implemented allocators, or arrays of structs with differently typed fields) to be correctly tracked. However, `dfec`'s interest in types is only so it can call constructors, destructors, and copy functions, and per-block tracking is sufficient for this. Some run-time type errors could be detected - for instance, using `memcpy()` to copy a region of type A into a region of type B. However, since `dfec` wants to be as transparent as possible to the user, in this case it simply updates the destination region's `dest` and `copy` function pointers with those of the source region, instead of issuing an error about the difference.

Chapter 7

Conclusion

This thesis has shown the design, implementation, and effectiveness of a C/C++ front end for the Daikon system, consisting of a source-to-source transformation utility and a runtime library to be linked into the instrumented source.

Testing, use, and the gradual evolution of the system has shown that augmenting C code while maintaining coherency (i.e., without disrupting the original semantics of the program) is extremely hard to do. C is a very low-level language, and allows the programmer to perform operations (like reinterpret casts) that can break any additional code linked into the system. Additionally, the C language does not provide protection against array overruns or other out of bounds accesses like Java does.

In spite of this, safeguards can be implemented that dynamically enforce safety. Adding in bounds-checking to a program not only alerts the user to a potential problem, but could actually increase the safety of the program, once fixed-length array padding is implemented — corner cases that used to have undefined behavior will now be well-defined, and won't have a possibility of crashing the program, where the original uninstrumented source may have crashed.

User feedback suggests, however, that any behavior that diverges from the behavior of the uninstrumented source is unacceptable, whether it informs the user about properties of the code in question or not. `dfec` is part of a system used to detect invariants, and is not a buffer overrun detector. Unfortunately, this puts `dfec` in the position of attempting to emulate undefined behavior, but progress can be made by carefully choosing the behavior that is the most tolerant of the running program.

Uninstrumented functions from `libc` pose a unique challenge in that they cannot be rewritten by the instrumenter. However, they are well-defined semantically, and can be assumed to be well-behaved, to a greater degree than user code can be. Additional information can be gleaned from the semantics of these functions, which can aid `dfec` in the discovery of more valid user data.

In summary, implementing a front end for Daikon consists of an inherent tradeoff between collecting as much data as possible from the running program, while ensuring (most importantly) that the program behavior is never changed. By playing it safe, keeping track of boundaries, and enhancing the picture by taking advantage of knowledge we have about the behavior of external functions, we can collect as much data as possible to export to Daikon proper for detection of quality invariants.

Bibliography

[Cah90] Conor P. Cahill. debug_malloc. http://sources.isc.org/devel/memleak/debug_malloc.txt, October 1990.

[EDG00] Edison Design Group. *C++ Front End Internal Documentation*, version 2.45 edition, March 2000. http://www.edg.com.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley, Reading, MA, 1995.

[Gui] Ron Guilmette. protoize: GNU Free Software Directory. http://www.gnu.org/directory/protoize.html.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proceedings of the Winter Usenix Conf.*, January 1992.

[HME02] Michael Harder, Benjamin Morse, and Michael D. Ernst. Improving test suites via generated specifications, February 1, 2002.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Software Series. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[LYHR01] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering*, Genova, Italy, 2-6 April 2001.

[Mey92] Scott Meyers. *Effective C++: 50 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, 1992.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th International Conference on Software Engineering*, pages 338–348, Boston, MA, May 1997.

[Plu99] Colin Plumb. MD5 Command Line Message Digest Utility. http://www.fourmilab.ch/md5, January 1999.

[PW00] Scott M. Pike and Bruce W. Weide. Checkmate: Cornering C++ dynamic memory errors with checked pointers. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, Austin, TX USA, March 7-12, 2000.

[RD01] Vincent Rijmen and Joan Daemen. The Rijndael Block Cipher. http://csrc.nist.gov/encryption/aes/rijndael/, February 2001.

[RH98] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24(6):401–419, June 1998.

[Sew] Julian Seward. Valgrind, an open-source memory debugger for x86-gnu/linux.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, special edition, 2000.