Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2015-036 December 27, 2015

Filtered Iterators For Safe and Robust Programs in RIFL

Jiasi Shen and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Filtered Iterators For Safe and Robust Programs in RIFL

Jiasi Shen
MIT CSAIL
jiasi@csail.mit.edu

Martin Rinard
MIT CSAIL
rinard@csail.mit.edu

Abstract

We present a new language construct, *filtered iterators*, for safe and robust input processing. Filtered iterators are designed to eliminate many common input-processing errors while enabling robust continued execution. The design is inspired by (a) observed common input-processing errors and (b) continued execution strategies that are implemented by developers fixing input validation errors. Filtered iterators decompose inputs into input units, atomically and automatically discarding units that trigger errors. Statistically significant results from a developer study highlight the difficulties that developers encounter when developing input-processing code using standard language constructs. These results also demonstrate the effectiveness of filtered iterators in eliminating many of these difficulties and enabling developers to produce safe and robust input-processing code.

1. Introduction

We present the design and evaluation of a new programming language construct, *filtered iterators*, for safe and robust input file processing. To target the problem of prevalent defects and security vulnerabilities in input-processing code, filtered iterators provide continued successful execution in the face of otherwise fatal or exploitable errors and emulate the effect of observed strategies from human developers fixing previous input-processing defects. Filtered iterators are designed to substantially simplify the development of safe and robust input-processing code.

We implement filtered iterators in the Robust Input Filtering Language (RIFL). We empirically evaluate the effectiveness of RIFL in a study in which matched pairs of developers develop input-processing programs with and without filtered iterators. The statistically significant results show that the use of filtered iterators produces code with fewer defects overall, fewer fatal defects, fewer crashes, and fewer conditional clauses (reflecting the reduced complexity of code). Moreover, the use of filtered iterators, by construction, completely eliminates many common sources of defects and security vulnerabilities.

Many past language design efforts have been driven by abstract philosophical goals such as simplicity [74, 79], orthogonality and consistency [76], generality [55], modularity [78], data abstraction [44], supporting analogies with the way people interact with the physical world [20, 32, 74], and enabling people to communicate with computers using math [9, 41, 50] or an approximation of English [60]. The design of filtered iterators, in contrast, was driven by the goal of eliminating a specific class of problems empirically evident by examining the development histories of existing software projects to understand the mistakes developers repeatedly make using existing languages. And instead of justifying the design with appeals to abstract philosophical principles, we justify the RIFL design via experiments that provide statistically significant empirical evidence for the benefits that it provides.

1.1 Main Concept

The key RIFL filtered iterator construct, the *inspect* construct, splits the input into input units, then iterates over the input units. Example input units include lines in a text file, packets in a network traffic capture file, user files in an archive, and images in an animation. Each iteration is an atomic (all-or-nothing) transaction that processes one input unit. RIFL discards *bad* input units (input units that trigger errors), enabling the program to continue processing the remaining input units as if the discarded input units had never existed.[1] RIFL therefore uses the program itself to find and filter bad input units. Direct benefits of filtered iterators include:

- **Survival:** Filtered iterators prevent program crashes and enable successful continued execution by automatically discarding bad input units that trigger fatal errors or target certain classes of security vulnerabilities.
- **Simplified Development:** Filtered iterators simplify software development by eliminating the need for many explicit checks and error-handling code. Developers can simply omit any check—and the associated error-handling code—that is otherwise required to prevent the program from crashing. Atomic execution also eliminates resource leaks that would otherwise occur when aborted subcomputations neglect to appropriately reclaim allocated resources [51].

[1] To help developers debug their programs, the RIFL runtime generates a log that contains all errors that occurred during execution.

- **Simplified Code:** The elimination of explicit checks and error-handling code simplifies the resulting implementation, enhancing the clarity and maintainability of the resulting program.
- **Fast Prototyping:** RIFL enables developers to write the main program logic quickly, without worrying about uncommon input units that may trigger errors. The RIFL runtime produces outputs for the input units that the program can process successfully.

1.2 Design Rationale

Input Processing Defects: RIFL targets input-processing defects, in part, because they occur frequently in practice and often have serious consequences such as security vulnerabilities and program failures [1, 10, 46, 46, 48, 49, 61, 62, 68, 72, 81]. Many of these defects correspond to uncommon corner cases that developers simply overlook. The consequences can be especially severe for programs exposed to potentially malicious inputs that purposefully target defects that well formed or anticipated inputs do not expose.

Continued Execution: RIFL's focus on continued execution by discarding bad input units is motivated, in part, by the empirical observation that human developers often take precisely this approach when fixing input-processing defects. Specifically, many human bug fixes for null dereference and divide-by-zero errors do not throw an exception or exit the program. They instead nullify the computation associated with the current input unit and continue on to process subsequent input units [46, 48]. For example, fixes to errors CVE-2013-2483, CVE-2012-4286, CVE-2012-4285, CVE-2012-1143, CVE-2012-4507, and CVE-2011-4153 in the CVE database [1] all exhibit this pattern. There are a variety of situations in which this continued execution provides important advantages:

- **Useful Partial Results:** For many computations, results computed from a subset of the input units are far preferable to a computation that simply terminates when it encounters a bad input unit [48]. This property may be particularly relevant when the input file contains multiple largely independent input units, such as multiple network packets in a network capture file [6, 23] or multiple images in an image file [3, 11, 37]. Many of the bug fixes discussed above are specifically designed to enable the program to provide useful partial results in the face of bad input units [48]. Systems for approximate scientific and big data computing often provide mechanisms that increase robustness by discarding bad computations [22, 56]. These mechanisms enable programs to deliver useful partial results even if the program cannot successfully process all input units [22, 56].
- **Safety Critical Systems:** Terminating a safety critical system can have catastrophic results. The alternative of terminating and attempting to reboot can be impractical: (a) A bad input unit may persist in a file that the system reads when it reboots, preventing the system from successfully rebooting. Such an error was present in the Center TRACON Automation System (CTAS) air-traffic control software [25]. (b) The time required to reboot, reacquire data, and re-calibrate the system may be unacceptable. This is the case for the CTAS air traffic control system. (c) It may be too difficult to recalculate critical values. This reason was explicitly cited in the Ariane-5 disaster report [29]. In these situations, continued execution, even with potentially reduced functionality, is far preferable to termination.

Many language implementations [28, 34, 59] check for conditions such as out of bounds accesses and null dereferences. If the check fires, the implementations typically either terminate the computation [28, 59] or throw an exception [34], which also typically terminates the computation when the exception propagates up to the default handler. In contrast to these mechanisms that terminate the program, RIFL's continued execution is designed to emulate the bug fixes that human developers produce in practice to maximize program utility via robust continued execution [48].

1.3 Contributions

This paper makes the following contributions.

- **Filtered Iterators:** We present filtered iterators, a novel control structure that partitions input files into input units and discards bad input units atomically and automatically. The goal is to enable robust continued execution without the need to consider or develop code that handles error conditions that uncommon corner-case inputs would trigger.
- **Empirical Evaluation:** We present results from an empirical evaluation of filtered iterators in RIFL. This evaluation is based on experiments that use paired human developers to test hypotheses regarding the ability of RIFL to deliver smaller, more robust programs with fewer defects.
  The statistically significant results highlight (a) the difficulties that developers face when developing input-processing code using standard constructs and (b) the effectiveness of filtered iterators in eliminating many of these difficulties. In our study, programs from developers using filtered iterators had a total of six defects, none fatal. The paired control group, in contrast, produced programs with a total of 33 defects, 18 fatal, with every program containing at least two fatal defects. We verified the exercisable presence of every defect—with the exception of the defects related to memory allocation failures and memory leaks, which we identified by a code analysis—by developing an input file that triggered the defect.
  We also briefly discuss our experience developing RIFL programs for a variety of input formats including the PCAP, PNG, ZIP, JSON, and OBJ formats.

```
Img1 2 2 2 1234
Img2 2 4 4 1234567890123456
CharS b 2 2 1234
CharPix 2 2 2 12a4
BufOvfVeryLongName 2 2 2 1234
2 2 2 1234
Div0S 0 2 2 1234
Div0H 2 0 2
HeapOvf 2 60000 1000 1234
BufOvfInt 2 16 268435457 12345678901234567890
Img3 2 1 2 12
Img4 3 3 4 123456789012
```

Figure 1: Example input for thumbnail programs

This paper advocates an empirical approach to programming language design. The field has matured to the point in which the specific problems that developers face, and the characteristics of the fixes they apply in response to these problems, are now becoming clear with experience. Designing constructs with the benefit of this experience, and then experimentally evaluating the success or failure of these constructs in ameliorating the targeted problems, can be one productive way to design new language constructs.

2. Example

A key principle of RIFL is to encourage writing only common-case code and handling errors implicitly with filtered iterators. To illustrate this idea, we contrast two implementations of a bitmap thumbnail generator program. The first implementation uses conventional language constructs to handle errors *explicitly*. The second implementation uses RIFL filtered iterators to handle errors *implicitly*.

Program Functionality: The thumbnail program uses the following algorithm to average the values of neighboring pixels. Given a scaling factor s, the thumbnail for an image of height h and width w has height $\lfloor h/s \rfloor$ and width $\lfloor w/s \rfloor$. The value of the pixel in row i and column j of the thumbnail is the floor of the average of the values of all pixels in the s^2 square area between rows $i \cdot s \ldots (i \cdot s + s - 1)$ and columns $j \cdot s \ldots (j \cdot s + s - 1)$ of the original image.

Program Input: Figure 1 presents an example input file. The input file contains lines that describe original images. Each line describes an original image with the following components, separated by a single space character: (a) A string that describes the image name. This string contains no space or newline characters and is 1–10 characters long. (b) An integer, s, that represents the scaling factor. (c) An integer, h, that represents the original image height. (d) An integer, w, that represents the original image width. (e) $h \cdot w$ consecutive digits that each represents the color of a pixel in the original image, ranging from '`0`'...'`9`'. The pixels represent the image by forming a matrix of h rows and w columns. We order the pixels as follows: inside each row, pixels are ordered from left to right; the rows are each grouped together and ordered from top to bottom.

```
Img1 2
Img2 3543
Img3
Img4 3
```

Figure 2: Example output for thumbnail programs

Program Output: Figure 2 presents the example output. Each line describes a thumbnail image with the following components, separated by a single space character: (a) The image name. (b) $\lfloor h/s \rfloor \lfloor w/s \rfloor$ digits that each represents the value of a pixel in the thumbnail.

Robustness: Like other image file formats [3, 11, 37], the input files in our example may contain multiple images. To maximize the utility of the program, an image that triggers errors should not prevent the program from generating thumbnails for other images.

The example input in Figure 1 contains invalid lines that do not conform to the input specification. Images `CharS` and `CharPix` contain illegal characters in the scaling factor and the pixels, respectively. Image `BufOvfVeryLongName`'s name is too long. In the line that follows, the image name is missing. Image `Div0S`'s scaling factor is zero, which is undefined for thumbnail generators.

There are other lines in the example input that satisfy the specification but may still trigger errors in implementations with defects. Image `Div0H` contains an empty image. Image `HeapOvf`'s dimensions may be too large to fit the entire image in the heap memory. Image `BufOvfInt`'s dimensions are so large that the multiplication of the image height and width overflows a 32-bit integer.

The example output in Figure 2 contains thumbnails for all images that do not trigger errors.

2.1 Conventional Version

Figure 3 presents a conventional implementation of the thumbnail generator that handles all errors explicitly. Language keywords are in **bold**. The control structure that loops through images is underlined. Error-handling code is highlighted with colored text, according to the following importance category:

- **Vital** code that prevents crashes is in red.
- **Service** code that cleans up resources is in brown.
- **Integrity** code that prevents input misinterpretation is in blue.
- **Common** code that is part of common engineering practice is in green.

Table 1 enumerates all the error-handling code in this implementation. Each row describes a piece of code. The columns are as follows: the identifier number (Id), the location (Lines), the importance category (Importance), the functionality (Purpose), and the number of conditional clauses and unconditional statements separated with a slash sign "/" (Count).

Table 1: Error-handling code in the conventional thumbnail program. Each pair of numbers separated by a slash "/" are the numbers of conditional clauses and of unconditional statements.

Id	Lines	Importance	Purpose	Count
1	2–5	integrity	flush trailing bytes for current unit (helper function)	1 / 4
2	10	integrity	avoid mixing bad units with following units	0 / 1
3	11	integrity	identify bad units	0 / 1
4	18	common	exit on invalid input files	1 / 1
5	20	vital	avoid null array accesses	1 / 1
6	22 (`bad=0;`)	integrity	default to good units	0 / 1
7	24 (`x<0`)	integrity	identify incomplete units at the ends of files	1 / 0
8	24 (`x=='\n'`)	integrity	avoid mixing bad units with following units	1 / 0
9	24 (`i>=10`)	vital	avoid out-of-bound array accesses	1 / 0
10	25	integrity	skip bad units	0 / 2
11	29 (`bad`)	integrity	skip bad units	1 / 2
12	32 (`s<=0`, `h<=0`)	vital	avoid divisions by zero	2 / 0
13	32 (`w<=0`)	integrity	identify malformed units	1 / 0
14	32 (`(h*w)/h!=w`)	vital	avoid integer overflows and out-of-bound array accesses	1 / 0
15	33	integrity	skip bad units	0 / 2
16	36	vital	avoid null array accesses	1 / 2
17	38 (`!bad`)	integrity	avoid mixing bad units with following units	1 / 0
18	42	integrity	identify bad units	0 / 2
19	47	integrity	avoid mixing bad units with following units	0 / 1
20	48 (`bad`)	integrity	skip bad units	1 / 1
21	48 (`free_ec(pix)`)	service	avoid memory leak on bad units	0 / 1

2.2 RIFL Version

None of this error-handling code is needed in the RIFL version of the thumbnail generator in Figure 4. This implementation has the same functionality as the conventional version. Unlike the conventional version, the RIFL version uses an `inspectt` loop to implicitly and atomically discard any input units that violate assertions or trigger other errors during an iteration.

RIFL's `inspectt` construct implements filtered iterators for text input formats. The (simplified) structure of an `inspectt` loop is as follows.

```
inspectt (e, f, du) { ... }
```

This loop iterates through input units in a text file `f` while expression `e` evaluates to "true". A delimiter `du` defines the boundaries between input units.

The RIFL runtime supports `inspectt` loops as follows. Like conventional `while` loops, an `inspectt` loop repeatedly executes a code block as long as a given condition holds. Unlike `while` loops, an `inspectt` loop additionally flushes inputs based on the delimiters. The `inspectt` loop avoids all detectable errors by skipping bad input units. RIFL processes each input unit *atomically* in each iteration, so that the updates from processing each input unit either commit or completely abort.

Eliminated Checks: We next discuss the checks and statements that are required in the conventional version but are redundant in the RIFL version. Checks #9, #12, #14, and #16, which are vital in the conventional version, are unnecessary here because the RIFL runtime implicitly triggers an error on failed heap allocations, out-of-bound array accesses, and divisions by zero. The `inspectt` loop starting from line 13 implicitly handles all these errors by discarding bad input units atomically. Check #5 is also unnecessary because a failed heap allocation on line 12 aborts the program immediately. Specifically, if the conventional version omits any of the vital error-handling code, the program would crash from bad inputs that trigger errors:

- **Out-of-bound array access:** Without check #9, the conventional version writes beyond array `name` on line 27 when an image name is longer than the array size.
- **Integer overflow and out-of-bound array access:** Without check #14, the conventional version writes beyond array `pix` on line 43 with carefully chosen image dimensions that cause the multiplication on line 35 to overflow the integer representation. This integer overflow can cause the program to allocate an array `pix` that is smaller than expected.
- **Division by zero:** Without check #12, the conventional version triggers division-by-zero errors on line 32 or inside the code for line 50 when an image contains a zero height or a zero scaling factor.
- **Null array access:** Without check #16, the conventional version writes to a null array on line 43 when the heap allocation fails on line 35. Without check #5, the conventional version writes to a null array on line 27 when the memory allocation on line 19 fails.

Apart from freeing the developers from having to write otherwise vital error-handling code, RIFL also eliminates the need for other important code in the conventional version. Statement #21 is unnecessary in the RIFL version be-

```
1  f=opent("images.txt");
2  func flush (x) {
3    while (x>=0 && x!='\n') { x=read(f); }
4    return 0;
5  }
6  func rdint () {
7    x=0; dgt=read(f);
8    while (dgt!=' ') {
9      if (dgt<'0' || dgt>'9') {
10       _=seek(f,pos(f)-1);
11       return -1;
12     }
13     x=x*10+(dgt-'0'); dgt=read(f);
14   }
15   return x;
16 }
17 main {
18   if (!valid(f)) { exit(1); }
19   name=malloc(10);
20   if (!valid(name)) { exit(1); }
21   while (!end(f)) {
22     bad=0; i=0; x=read(f);
23     while (x!=' ') {
24       if (x<0 || x=='\n' || i>=10) {
25         bad=1; break;
26       }
27       name[i]=x; i=i+1; x=read(f);
28     }
29     if (i<=0 || bad) { _=flush(x); continue; }
30     while (i<10) { name[i]=0; i=i+1; }
31     s=rdint(); h=rdint(); w=rdint();
32     if (s<=0 || h<=0 || w<=0 || (h*w)/h!=w) {
33       _=flush(0); continue;
34     }
35     pix=malloc(h*w);
36     if (!valid(pix)) { _=flush(0); continue; }
37     i=0;
38     while (i<h && !bad) {
39       j=0;
40       while (j<w) {
41         x=read(f);
42         if (x<'0' || x>'9') { bad=1; break; }
43         pix[i*w+j]=x; j=j+1;
44       }
45       i=i+1;
46     }
47     _=flush(x);
48     if (bad) { free(pix); continue; }
49     print(name); print(' ');
50     // « compute and print averages, code omitted »
51     print('\n'); free(pix);
52   }
53   free(name);
54   return 0;
55 }
```

Figure 3: A conventional thumbnail implementation

cause the `inspectt` loop discards bad input units atomically, preventing a memory leak in this situation. Combining error handling with iterators, RIFL also eliminates the need to explicitly maintain the integrity of input units as in (a) statements #1, #2, #3, #6, #10, #15, #18, and #19, or (b) checks #7, #8, #11, #13, #17, and #20. On the other hand, the conventional version would suffer from omitting these important code:

- **Memory leak:** Without statement #21, the conventional version leaks memory on images with corrupted pixel values. Memory leaks degrade the quality of service and may result in allocation failures.

```
1  f=opent("images.txt");
2  func rdint () {
3    x=0; dgt=read(f);
4    while (dgt!=' ') {
5      assert(dgt>='0' && dgt<='9');
6      x=x*10+(dgt-'0'); dgt=read(f);
7    }
8    return x;
9  }
10
11 main {
12   name=malloc(10);
13   inspectt (!end(f),f,'\n') {
14     i=0; x=read(f);
15     while (x!=' ') {
16       name[i]=x; i=i+1; x=read(f);
17     }
18     assert(i>=1);
19     while (i<10) { name[i]=0; i=i+1; }
20     s=rdint(); h=rdint(); w=rdint();
21     pix=malloc(h*w);
22     i=0;
23     while (i<h) {
24       j=0;
25       while (j<w) {
26         x=read(f);
27         assert(x>='0' && x<='9');
28         pix[i*w+j]=x; j=j+1;
29       }
30       i=i+1;
31     }
32     print(name); print(' ');
33     // « compute and print averages, code omitted »
34     print('\n'); free(pix);
35   }
36   free(name);
37   return 0;
38 }
```

Figure 4: A RIFL thumbnail implementation

- **Input misinterpretation:** Other code is necessary for the integrity of input units on errors. Code #1, #2, #8, #10, #15, #17, #18, and #19 isolate bad input units from other input units. Code #3, #6, #7, #11, #13, and #20 detect bad input units.
- **Common practice:** Code #4 checks whether the opening operation on line 1 succeeded. Developers should check this error early, even if missing this check may not cause immediate crashes.

The RIFL version is shorter and significantly simpler than the conventional version. This fact is consistent with the intuition that filtered iterators can simplify the implementation of robust programs.

3. The RIFL Language

RIFL is a standard imperative language augmented with two inspect loop constructs: the `inspectt` construct for text files and the `inspectb` construct for binary files.

3.1 Syntax

Figure 5 presents the syntax of RIFL. The structure of inspect loops for text inputs is

```
inspectt (e, f, du, ds) { ... }
```

Prog := Stmt | `func` $q(x)${Stmt; `return` y}; Prog

Exp := n | x | Exp op Exp | a[Exp] | `valid`(a) | `end`(f) | `pos`(f)

Stmt := x = Exp | a = `malloc`(Exp) | `free`(a) | a[Exp] = Exp
| x = q(Exp) | Stmt; Stmt | `if`(Exp){Stmt}`else`{Stmt}
| `while`(Exp){Stmt} | `inspectt`(Exp, f, d_u, d_s){Stmt}
| `inspectb`(Exp, f, Exp, Exp, Exp){Stmt}
| f = `opent`(str) | f = `openb`(str) | `seek`(f, Exp)
| x = `read`(f) | `assert`(Exp)

$x, y \in IVar$ — $q \in$ function names

$a, d_u, d_s \in AVar$ — $n \in Int$

$f \in FVar$ — $str \in String$

Figure 5: Abstract syntax

which, while expression `e` evaluates to "true", iterates through input units in a text file `f`. Each loop iteration may access an input unit that consists of the contents of file `f` up to the end-of-unit delimiters specified in `du`. The loop terminates when `e` evaluates to "false", when the program reads the end-of-sequence delimiters specified in `ds`, or when the program reaches the end of file `f`.

The structure of inspect loops for binary inputs is

```
inspectb (e, f, o, w, c) { ... }
```

which, while expression `e` evaluates to "true", iterates through input units in a binary file `f`. Each loop iteration may access an input unit that consists of the contents of file `f` up to a cutoff position as specified by the parameters `o`, `w`, and `c`. The loop terminates when `e` evaluates to "false", when the program reaches the end of an outer-level input unit, or when the program reaches the end of file `f`. The cutoff position for each input unit is computed as follows. Before each loop iteration, the RIFL runtime identifies the length field in file `f` using the offset `o` and the width `w`. It extracts the value of the length field according to the endianness that the developer specifies when opening file `f`. The RIFL runtime then precomputes a cutoff position of the current input unit by summing up the current file offset, the value of the length field, and the extra length `c`.

Atomic Execution and Filtering: In addition to iterating, inspect loops also handle errors by filtering out bad input units. The RIFL runtime includes checks for implicit error conditions such as out-of-bounds accesses, null pointer dereferences, divisions by zero, and memory allocation failures. When an iteration of an inspect loop triggers an error, either because of a failed RIFL check or because of an explicit assertion failure, the RIFL runtime (a) identifies the bad input unit, (b) advances the file pointer past the bad input unit, (c) restores the rest of the program's state, and (d) restarts the program execution at the start of the next input unit (unless the bad input unit is the last input unit in the file).

Nesting and Recursion: Besides sequential input units, inspect loops can also process complex input structures with nesting and recursion. For example, the Comma Separated Values (CSV) format [63] delimits rows with newline characters and delimits fields inside each row with comma characters. Rows are larger input units and fields are smaller input units. A program that processes CSV files may contain two nested inspect loops, where the outer loop uses the newline delimiter and the inner loop uses the comma delimiter. As another example, the JavaScript Object Notation (JSON) [12] reuses the right brace character as the delimiter for objects across all nesting layers. Objects are input units. A program that processes the JSON format may contain a recursive function where an inspect loop uses the right brace delimiter and parses inner input units by calling the function recursively.

3.2 Operational Semantics

We developed a RIFL interpreter based on the following operational semantics. RIFL's operational semantics uses the following domain. A state

$$\sigma \in State = Stack \times Heap \times Files \times Disk$$

contains information about the stack memory, the heap memory, the status of opened files and the disk. The stack maps variables to values, which can be integers, file handlers or memory addresses. The heap maps memory addresses to array contents. The file status maps file handlers to file descriptors. The disk maps file names to file contents.

$$Stack = Var \to Value$$
$$Heap = Addr \to Data \times Size$$
$$Files = FHndl \to FDesc$$
$$Disk = FName \to Data \times Size$$

where $Var = IVar \cup FVar \cup AVar$, $Value = Int \cup FHndl \cup Addr$, $Data = Offs \to Int$, and $Size = Offs = Int$.

A file descriptor $fd \in FDesc = FName \times Offs \times SOU \times UDesc$ describes the current status of reading an input file, including the file name, the current offset into the file, the starting offset of the current input unit, and an input unit descriptor. $FName = String$, $SOU = Int$. An input unit descriptor $ud \in UDesc = Delim \cup Cutoff$ describes the delimiters in use for text files and the cutoff positions for binary files.

For text files, a delimiter definition $dlm \in Delim = EOU \times EOS \times OSD$ describes three sets of delimiters that identify the boundaries between input units: $dlm = \langle eou, eos, osd \rangle$ where $eou \in EOU = \mathcal{P}(Int)$ is the set of end-of-unit delimiters, $eos \in EOS = \mathcal{P}(Int)$ is the set of end-of-sequence delimiters, and $osd \in osd = \mathcal{P}(Int)$ is the set of outside delimiters that serve nested input units. The next delimiter in the input file, whether it is one in $eou \cup eos \cup osd$ or the end of the file, marks the end of

the current input unit. The set of outside delimiters osd updates at runtime as follows. For single-layer `inspectt` loops and the outermost `inspectt` loops in nested structures, $osd = \varnothing$. For inner `inspectt` loops, osd includes all delimiters in $eou \cup eos$ for all the outer `inspectt` layers, except for those that also appear in $eou \cup eos$ of the current layer. This exception is useful for input formats that reuse delimiters across layers, such as JSON. However, the developer should be careful about reusing delimiters across the hierarchy. Reuse can make delimiters ambiguous, which may cause the program to misinterpret the input structures in the face of delimiter corruptions.

For binary files, a cutoff position $cut \in Cutoff = Int$ describes where the current input unit ends. This value also updates at runtime according to nesting `inspectb` layers.

The relation $\langle e, \sigma\rangle \Downarrow_e \mu$ denotes that evaluating the expression e in state σ yields the result $\mu \in Int \cup \{\text{err}\}$. A result $\mu \in Int$ indicates that the evaluation is successful and that the numerical result is μ. A result $\mu = \text{err}$ indicates that the evaluation fails, which would then trigger an error in the surrounding statement.

The relation $\langle s, \sigma\rangle \Downarrow_s \xi$ denotes that executing the statement s in the state σ yields the output configuration $\xi \in State \times \{\text{ok}, \text{bad}\}$. An output configuration $\xi = \langle \sigma', \text{ok}\rangle$ indicates that the program execution is successful and that the resulting state is σ'. An output configuration $\xi = \langle \sigma', \text{bad}\rangle$ indicates that the program execution triggered an error and that the latest reasonable program state is σ'. In this case, the RIFL runtime would report the error to the surrounding inspect loop which would resolve the problem.

3.2.1 Basic Operations

Figures 6–11 present basic operations. Figure 6 presents the semantics for simple expressions. Arithmetic errors and invalid array reads trigger errors in the surrounding statement (iop-bad, ard-null, ard-out). The `valid` predicate tests whether an array variable is not null (avalid-t, avalid-f). Figure 7 presents the semantics for simple assignments. When assigning a bad expression to a variable, rule (vwr-bad) treats the statement as a no-op and reports the error. Figure 8 presents the semantics for arrays. A successful `malloc` statement allocates a space of the specified size in the heap, initializes all the elements to 0, and sets the array variable to the heap address (malloc-ok). A successful `free` statement deallocates the space from the heap and resets the array variable to null (free-ok). A successful assignment to an array element changes the specified element in the heap (awr-ok). On bad expressions, invalid `malloc` parameters, allocation failures, null array frees, null array accesses, or out-of-bounds array writes, rules (malloc-bad, awr-bad, malloc-neg, malloc-ovf, free-null, awr-null, awr-out) treat the statement as a no-op and reports the error. Figure 9 presents the semantics for function calls using the following helper functions:

$$\frac{}{\langle n, \sigma\rangle \Downarrow_e n} \quad \text{(int)}$$

$$\frac{}{\langle x, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e \sigma_S(x)} \quad \text{(ivar)}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow_e u_1 \quad \langle e_2, \sigma\rangle \Downarrow_e u_2 \quad u_1 \text{ op } u_2 = v}{\langle e_1 \text{ op } e_2, \sigma\rangle \Downarrow_e v} \quad \text{(iop-ok)}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow_e u_1 \quad \langle e_2, \sigma\rangle \Downarrow_e u_2 \quad u_1 \text{ op } u_2 = \bot}{\langle e_1 \text{ op } e_2, \sigma\rangle \Downarrow_e \text{err}} \quad \text{(iop-bad)}$$

$$\frac{\sigma_S(a) = \text{null}}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e \text{err}} \quad \text{(ard-null)}$$

$$\frac{\sigma_S(a) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n\rangle \quad u < 0 \vee u \geq n}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e \text{err}} \quad \text{(ard-out)}$$

$$\frac{\sigma_S(a) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n\rangle \quad 0 \leq u < n}{\langle a[e], \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e \gamma(u)} \quad \text{(ard-ok)}$$

$$\frac{\sigma_S(a) \neq \text{null}}{\langle \texttt{valid}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e \text{true}} \quad \text{(avalid-t)}$$

$$\frac{\sigma_S(a) = \text{null}}{\langle \texttt{valid}(a), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e \text{false}} \quad \text{(avalid-f)}$$

Figure 6: Semantics for simple expressions

$$\frac{\langle e, \sigma\rangle \Downarrow_e \text{err}}{\langle x = e, \sigma\rangle \Downarrow_s \langle \sigma, \text{bad}\rangle} \quad \text{(vwr-bad)}$$

$$\frac{\langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_e u}{\langle x = e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto u], \sigma_H, \sigma_F, \sigma_D\rangle, \text{ok}\rangle} \quad \text{(vwr-ok)}$$

Figure 7: Semantics for assignments

$$\frac{\langle s_1, \sigma\rangle \Downarrow_s \langle \sigma', \text{bad}\rangle}{\langle s_1; s_2, \sigma\rangle \Downarrow_s \langle \sigma', \text{bad}\rangle} \quad \text{(seq-bad)}$$

$$\frac{\langle s_1, \sigma\rangle \Downarrow_s \langle \sigma', \text{ok}\rangle \quad \langle s_2, \sigma'\rangle \Downarrow_s \xi}{\langle s_1; s_2, \sigma\rangle \Downarrow_s \xi} \quad \text{(seq-prgr)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \text{err}}{\langle \texttt{if}(e)\{s_1\}\texttt{else}\{s_2\}, \sigma\rangle \Downarrow_s \langle \sigma, \text{bad}\rangle} \quad \text{(if-bad)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \text{true} \quad \langle s_1, \sigma\rangle \Downarrow_s \xi}{\langle \texttt{if}(e)\{s_1\}\texttt{else}\{s_2\}, \sigma\rangle \Downarrow_s \xi} \quad \text{(if-t)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \text{false} \quad \langle s_2, \sigma\rangle \Downarrow_s \xi}{\langle \texttt{if}(e)\{s_1\}\texttt{else}\{s_2\}, \sigma\rangle \Downarrow_s \xi} \quad \text{(if-f)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \text{err}}{\langle \texttt{while}(e)\{s\}, \sigma\rangle \Downarrow_s \langle \sigma, \text{bad}\rangle} \quad \text{(while-bad)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \text{false}}{\langle \texttt{while}(e)\{s\}, \sigma\rangle \Downarrow_s \langle \sigma, \text{ok}\rangle} \quad \text{(while-end)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \text{true} \quad \langle s, \sigma\rangle \Downarrow_s \langle \sigma', \text{bad}\rangle}{\langle \texttt{while}(e)\{s\}, \sigma\rangle \Downarrow_s \langle \sigma', \text{bad}\rangle} \quad \text{(while-body)}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_e \text{true} \quad \langle s, \sigma\rangle \Downarrow_s \langle \sigma', \text{ok}\rangle \quad \langle \texttt{while}(e)\{s\}, \sigma'\rangle \Downarrow_s \xi}{\langle \texttt{while}(e)\{s\}, \sigma\rangle \Downarrow_s \xi} \quad \text{(while-prgr)}$$

Figure 10: Semantics for basic control structures

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle a = \texttt{malloc}(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle}\ \text{(malloc-bad)} \qquad \frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} u \quad u \le 0}{\langle a = \texttt{malloc}(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle}\ \text{(malloc-neg)} \qquad \frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} u \quad u > 0 \quad \text{heap allocate}(u) = \bot}{\langle a = \texttt{malloc}(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle}\ \text{(malloc-ovf)}$$

$$\frac{\langle e,\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u \quad u > 0 \quad \text{heap allocate}(u) = \mathrm{addr}}{\langle a = \texttt{malloc}(e),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S[a \mapsto \mathrm{addr}], \sigma_H[\mathrm{addr} \mapsto \langle[0 \mapsto 0, 1 \mapsto 0, \ldots, u-1 \mapsto 0], u\rangle], \sigma_F, \sigma_D\rangle, \mathrm{ok}\rangle}\ \text{(malloc-ok)}$$

$$\frac{\sigma_S(a) = \mathrm{null}}{\langle \texttt{free}(a), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \mathrm{bad}\rangle}\ \text{(free-null)}$$

$$\frac{\sigma_S(a) \neq \mathrm{null}}{\langle \texttt{free}(a), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S[a \mapsto \mathrm{null}], \sigma_H[\sigma_S(a) \mapsto \bot], \sigma_F, \sigma_D\rangle, \mathrm{ok}\rangle}\ \text{(free-ok)}$$

$$\frac{\sigma_S(a) = \mathrm{null}}{\langle a[e_1] = e_2, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \mathrm{bad}\rangle}\ \text{(awr-null)}$$

$$\frac{\sigma_S(a) \neq \mathrm{null} \quad \langle e_1, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} \mathrm{err} \vee \langle e_2, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle a[e_1] = e_2, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle}\ \text{(awr-bad)}$$

$$\frac{\sigma_S(a) \neq \mathrm{null} \quad \langle e_1, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n\rangle \quad u < 0 \vee u \ge n}{\langle a[e_1] = e_2, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \mathrm{bad}\rangle}\ \text{(awr-out)}$$

$$\frac{\sigma_S(a) \neq \mathrm{null} \quad \langle e_1, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u_1 \quad \sigma_H(\sigma_S(a)) = \langle \gamma, n\rangle \quad 0 \le u_1 < n \quad \langle e_2, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u_2}{\langle a[e_1] = e_2, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H[\sigma_S(a) \mapsto \langle \gamma[u_1 \mapsto u_2], n\rangle], \sigma_F, \sigma_D\rangle, \mathrm{ok}\rangle}\ \text{(awr-ok)}$$

Figure 8: Semantics for arrays

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle x = q(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle}\ \text{(fn-arg)} \qquad \frac{\text{stack allocate}(fr(q)) = \bot}{\langle x = q(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle}\ \text{(fn-ovf)}$$

$$\frac{\text{stack allocate}(fr(q)) \neq \bot \quad \langle e,\sigma\rangle \Downarrow_{\mathrm{e}} u \quad \langle body(q), \langle [arg(q) \mapsto u], \sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma'_S,\sigma'_H,\sigma'_F,\sigma'_D\rangle, \mathrm{bad}\rangle}{\langle x = q(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma_S,\sigma_H,\sigma'_F,\sigma_D\rangle, \mathrm{bad}\rangle}\ \text{(fn-body)}$$

$$\frac{\text{stack allocate}(fr(q)) \neq \bot \quad \langle e,\sigma\rangle \Downarrow_{\mathrm{e}} u \quad \langle body(q), \langle [arg(q) \mapsto u], \sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma'_S,\sigma'_H,\sigma'_F,\sigma'_D\rangle, \mathrm{ok}\rangle}{\langle x = q(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S[x \mapsto \sigma'_S(ret(q))], \sigma'_H,\sigma'_F,\sigma'_D\rangle, \mathrm{ok}\rangle}\ \text{(fn-prgr)}$$

Figure 9: Semantics for function calls

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle \texttt{assert}(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle}\ \text{(assert-bad)} \qquad \frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{true}}{\langle \texttt{assert}(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{ok}\rangle}\ \text{(assert-t)} \qquad \frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{false}}{\langle \texttt{assert}(e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{bad}\rangle}\ \text{(assert-f)}$$

Figure 11: Semantics for assertions

For each function definition $\texttt{func}\ q(x)\{s; \texttt{return}\ y\}$,
let $arg(q) \triangleq x$, $body(q) \triangleq s$, $ret(q) \triangleq y$, and
$fr(q) \triangleq$ size of q's stack frame.

A successful function call updates the global states and assigns the return value to the receiving variable (fn-prgr). If the argument uses a bad expression or if the stack overflows, rules (fn-arg, fn-ovf) treat the function call as a no-op and report the error. If an error occurs inside the function call, rule (fn-body) updates only the file descriptors and then reports the error. Figure 10 presents the semantics for basic control structures. When an error occurs, the program stops executing and reports the error (seq-bad, while-body). If an inspect loop surrounds these statements, this inspect loop would discard the updates in the current iteration and would restart with the remaining input. Figure 11 presents the semantics for assertions. True assertions are no-ops (assert-t); false assertions or bad expressions generate errors (assert-f, assert-bad).

3.2.2 Filtered Iterators

Figures 12–15 present the semantics for filtered iterators. An inspect loop automatically maintains the file descriptor and other program states, using delimiters for text files and length fields for binary files as follows.

Text input formats: Figures 12 and 13 present the semantics for filtered iterators for text input files, using the following helper functions:

$\Omega(a) \triangleq \{j \in Int \mid \exists i \in Int, \sigma_H(\sigma_S(a)) = \langle \gamma, n\rangle, \gamma(i) = j\}$
$(a \in AVar,\ a \neq \mathrm{null})$
returns the set of elements in array a.

$\Lambda'(l') \triangleq \underset{k \ge l'}{\mathrm{argmin}}\{k = n' \vee \gamma'(k) \in eou' \cup eos' \cup osd'\}$
$(l' = 0, 1, \ldots, n')$
returns the offset of the upcoming delimiter from offset l'.

$$\frac{\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{false}}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(inspt-end)}$$

$$\frac{\begin{array}{c}\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle\rangle \\ dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle\rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') \geq n' \vee \gamma'(\Lambda'(l')) \in osd' \end{array}}{\begin{array}{c}\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \\ \langle \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle str, \Lambda'(l'), sou, \langle eou, eos, osd \rangle\rangle], \sigma'_D \rangle, \text{ok} \rangle \end{array}} \quad \text{(inspt-osd)}$$

$$\frac{\begin{array}{c}\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle\rangle \\ dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle\rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in eos' \end{array}}{\begin{array}{c}\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \\ \langle \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle str, \Lambda'(l') + 1, sou, \langle eou, eos, osd \rangle\rangle], \sigma'_D \rangle, \text{ok} \rangle \end{array}} \quad \text{(inspt-eos)}$$

$$\frac{\begin{array}{c}\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle\rangle \\ dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle\rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in eou' \\ \langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle str, \Lambda'(l') + 1, sou, \langle eou, eos, osd \rangle\rangle], \sigma'_D \rangle\rangle \Downarrow_{\text{s}} \xi \end{array}}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \xi} \quad \text{(inspt-prgr)}$$

Figure 12: Semantics for filtered iterators—with good text inputs

$$\frac{\sigma_S(d_u) = \text{null} \vee \sigma_S(d_s) = \text{null}}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(inspt-null)}$$

$$\frac{\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{err}}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(inspt-bad)}$$

$$\frac{\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) \neq \varnothing}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(inspt-dupl)}$$

$$\frac{\begin{array}{c}\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle\rangle \\ dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{bad} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle\rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') \geq n' \vee \gamma'(\Lambda'(l')) \in osd' \end{array}}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, \Lambda'(l'), sou, \langle eou, eos, osd \rangle\rangle], \sigma_D \rangle, \text{ok} \rangle} \quad \text{(inspt-dsc-osd)}$$

$$\frac{\begin{array}{c}\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle\rangle \\ dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{bad} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle\rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in eos' \end{array}}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, \Lambda'(l') + 1, sou, \langle eou, eos, osd \rangle\rangle], \sigma_D \rangle, \text{ok} \rangle} \quad \text{(inspt-dsc-eos)}$$

$$\frac{\begin{array}{c}\sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle\rangle \\ dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle\rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{bad} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle\rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in eou' \\ \langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, \Lambda'(l') + 1, sou, \langle eou, eos, osd \rangle\rangle], \sigma_D \rangle\rangle \Downarrow_{\text{s}} \xi \end{array}}{\langle \texttt{inspectt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_{\text{s}} \xi} \quad \text{(inspt-dsc-eou)}$$

Figure 13: Semantics for filtered iterators—with bad text inputs

$$\frac{\sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad l = \Lambda(l)}{\langle \texttt{end}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} \text{true}} \quad \text{(end-t)}$$

$$\frac{\sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad l < \Lambda(l)}{\langle \texttt{end}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} \text{false}} \quad \text{(end-f)}$$

$$\frac{\sigma_F(\sigma_S(f)) = \langle str, l, sou, ud\rangle}{\langle \texttt{pos}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} l} \quad \text{(pos)}$$

Figure 16: Semantics for input file expressions

An `inspectt` loop updates the starting offset of current input unit, updates the delimiters in use, and advances the offset according to the boundaries of input units (inspt-prgr). The loop terminates if the predicate evaluates to "false" (inspt-end) or if the program reaches the end of the unit sequence (inspt-eos, inspt-osd). Situations that end a sequence include reaching one of the explicit d_s delimiters, reaching a delimiter from outer `inspectt` layers, and reaching the end of the file. An `inspectt` loop handles a bad input unit by advancing the offset past the bad unit, restoring all other program states, and recovering the execution (inspt-dsc-eou, inspt-dsc-eos, inspt-dsc-osd). The delimiter arrays and the loop predicate should be valid (inspt-null, inspt-bad). The two sets of delimiters $\Omega(d_u)$ and $\Omega(d_s)$ should not intersect (inspt-dupl).

Binary input formats: Figures 14 and 15 present the semantics for filtered iterators for binary input files, using the following helper function:

$$parseint(l, u) \triangleq \begin{cases} 0, & \text{if } u = 0 \\ \text{the integer value decoded from} \\ \text{bytes } \gamma(l), \ldots, \gamma(l+u-1), & \text{if } u = 1, 2, \ldots, n-l+1 \end{cases}$$

$(l = 0, 1, \ldots, n)$

returns the integer value decoded from the u bytes starting from offset l.

An `inspectb` loop updates the starting offset of current input unit, updates the cutoff position, and advances the offset according to the boundaries of input units (inspb-prgr). The loop terminates if the predicate evaluates to "false" (inspb-end) or if the program reaches the end of the unit sequence (inspb-eos). Situations that end a sequence include reaching the cutoff position of the surrounding `inspectb` layer, reaching a zero-length input unit, or reaching a negative length field. An `inspectb` loop handles a bad input unit by advancing the offset past the bad unit, restoring all other program states, and recovering the execution (inspb-dsc-eou, inspb-dsc-eos). The parameters that identify the length field should be nonnegative (inspb-neg). The loop predicate and parameters should be valid (inspb-bad).

3.2.3 Explicit File Operations

Figures 16–18 present explicit file operations using the following helper function:

$$\Lambda(l) \triangleq \begin{cases} \underset{k \geq l}{\text{argmin}}\{k = n \vee \\ \quad \gamma(k) \in eou \cup eos \cup osd\}, & \text{if } ud = \langle eou, eos, osd\rangle \in Delim \\ cut, & \text{if } ud = cut \in Cutoff \end{cases}$$

$(l = 0, 1, \ldots, n)$

returns the offset of the upcoming delimiter from offset l for text inputs, and returns the upcoming cutoff position for binary inputs.

Figure 16 presents the semantics for input file expressions. The `end` predicate uses the file size and the input unit descriptors from all the nested inspect loop layers to test whether the input file offset is at the end of the current input unit (end-t, end-f). The `pos` function returns the current offset of the file descriptor (pos). Figure 17 presents the semantics for opening input files. A successful `opent` statement sets the file variable to a fresh text file handler (opent-ok). A successful `openb` statement sets the file variable to a fresh binary file handler (openb-ok). Figure 18 presents the semantics for accessing input files. A successful `seek` statement sets the file offset to the specified position (sk-ok). A successful `read` operation assigns the current input byte to the receiving variable and advances the offset in the file descriptor (frd-ok). The new position for `seek` must be inside the current input unit (sk-l, sk-r). Likewise, the developer should not invoke `read` at a delimiter in a text file, at a cutoff position in a binary file, or at the end of the file (frd-out). Instead, they should use the `end` predicate to test before reading. To read multiple bytes into an array, the developer may write a loop that `read`s multiple times. While another operation that is dedicated for this purpose would be expressive, the rules for this operation are complicated and do not add much insight beyond handling array bounds and input unit boundaries.

4. Controlled Experiment

We evaluate the RIFL design with a controlled experiment that uses pairwise comparison [69, 80] to evaluate the use of filtered iterators in program development. We are interested in the following research questions:

- **RQ1:** Does `inspect` reduce program *defects*?
- **RQ2:** Does `inspect` increase program *survivals*?
- **RQ3:** Does `inspect` reduce *wrong outputs*?
- **RQ4:** Does `inspect` reduce *wrong outputs* even if we apply failure-oblivious computing [57] to control group programs?
- **RQ5:** Does `inspect` reduce program *complexity*?

4.1 Methodology

Experimental Design: We first defined an input processing task, specifically, a thumbnail generator for files that may contain multiple images, as in Section 2. We then recruited twelve subjects to write a program that performed the task. These subjects were drawn from the academic population at

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \text{false}}{\langle \texttt{inspectb}(e,f,o,w,c)\{s\},\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma, \mathrm{ok}\rangle} \quad \text{(inspb-end)}$$

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \text{true} \quad \langle o,\sigma\rangle \Downarrow_{\mathrm{e}} u_o \quad \langle w,\sigma\rangle \Downarrow_{\mathrm{e}} u_w \quad \langle c,\sigma\rangle \Downarrow_{\mathrm{e}} u_c \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \quad \sigma_F(\sigma_S(f)) = \langle str,l,sou,cut\rangle \\ \sigma_D(str) = \langle \gamma, n\rangle \quad u_o \geq 0 \quad u_w \geq 0 \quad parseint(l+u_o,u_w) = v \quad cut = l \vee u_o = u_w = u_c = 0 \vee v < 0\end{array}}{\langle \texttt{inspectb}(e,f,o,w,c)\{s\},\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F[\sigma_S(f)\mapsto\langle str,cut,sou,cut\rangle],\sigma_D\rangle,\mathrm{ok}\rangle} \quad \text{(inspb-eos)}$$

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \text{true} \quad \langle o,\sigma\rangle \Downarrow_{\mathrm{e}} u_o \quad \langle w,\sigma\rangle \Downarrow_{\mathrm{e}} u_w \quad \langle c,\sigma\rangle \Downarrow_{\mathrm{e}} u_c \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \quad \sigma_F(\sigma_S(f)) = \langle str,l,sou,cut\rangle \\ \sigma_D(str) = \langle \gamma, n\rangle \quad u_o \geq 0 \quad u_w \geq 0 \quad parseint(l+u_o,u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l+u_o+u_w+v+u_c = cut' \quad l < cut' \leq cut \\ \langle s,\langle \sigma_S,\sigma_H,\sigma_F[\sigma_S(f)\mapsto\langle str,l,l,cut'\rangle],\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma'_S,\sigma'_H,\sigma'_F,\sigma'_D\rangle,\mathrm{ok}\rangle \\ \langle \texttt{inspectb}(e,f,o,w,c)\{s\},\langle \sigma'_S,\sigma'_H,\sigma'_F[\sigma'_S(f)\mapsto\langle str,cut',sou,cut\rangle],\sigma'_D\rangle\rangle \Downarrow_{\mathrm{s}} \xi\end{array}}{\langle \texttt{inspectb}(e,f,o,w,c)\{s\},\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \xi} \quad \text{(inspb-prgr)}$$

Figure 14: Semantics for filtered iterators—with good binary inputs

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err} \vee \langle o,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err} \vee \langle w,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err} \vee \langle c,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle \texttt{inspectb}(e,f,o,w,c)\{s\},\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma,\mathrm{bad}\rangle} \quad \text{(inspb-bad)}$$

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \text{true} \quad \langle o,\sigma\rangle \Downarrow_{\mathrm{e}} u_o \quad \langle w,\sigma\rangle \Downarrow_{\mathrm{e}} u_w \quad \langle c,\sigma\rangle \Downarrow_{\mathrm{e}} u_c \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \\ \sigma_F(\sigma_S(f)) = \langle str,l,sou,cut\rangle \quad \sigma_D(str) = \langle \gamma,n\rangle \quad u_o < 0 \vee u_w < 0 \vee u_c < 0\end{array}}{\langle \texttt{inspectb}(e,f,o,w,c)\{s\},\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle,\mathrm{bad}\rangle} \quad \text{(inspb-neg)}$$

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \text{true} \quad \langle o,\sigma\rangle \Downarrow_{\mathrm{e}} u_o \quad \langle w,\sigma\rangle \Downarrow_{\mathrm{e}} u_w \quad \langle c,\sigma\rangle \Downarrow_{\mathrm{e}} u_c \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \\ \sigma_F(\sigma_S(f)) = \langle str,l,sou,cut\rangle \quad \sigma_D(str) = \langle \gamma,n\rangle \quad u_o \geq 0 \quad u_w \geq 0 \\ parseint(l+u_o,u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l+u_o+u_w+v+u_c = cut' \quad cut' > cut\end{array}}{\langle \texttt{inspectb}(e,f,o,w,c)\{s\},\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F[\sigma_S(f)\mapsto\langle str,cut,sou,cut\rangle],\sigma_D\rangle,\mathrm{ok}\rangle} \quad \text{(inspb-dsc-eos)}$$

$$\frac{\begin{array}{c}\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \text{true} \quad \langle o,\sigma\rangle \Downarrow_{\mathrm{e}} u_o \quad \langle w,\sigma\rangle \Downarrow_{\mathrm{e}} u_w \quad \langle c,\sigma\rangle \Downarrow_{\mathrm{e}} u_c \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \\ \sigma_F(\sigma_S(f)) = \langle str,l,sou,cut\rangle \quad \sigma_D(str) = \langle \gamma,n\rangle \quad u_o \geq 0 \quad u_w \geq 0 \\ parseint(l+u_o,u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l+u_o+u_w+v+u_c = cut' \quad l < cut' \leq cut \\ \langle s,\langle \sigma_S,\sigma_H,\sigma_F[\sigma_S(f)\mapsto\langle str,l,l,cut'\rangle],\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle \sigma',\mathrm{bad}\rangle \\ \langle \texttt{inspectb}(e,f,o,w,c)\{s\},\langle \sigma_S,\sigma_H,\sigma_F[\sigma_S(f)\mapsto\langle str,cut',sou,cut\rangle],\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \xi\end{array}}{\langle \texttt{inspectb}(e,f,o,w,c)\{s\},\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \xi} \quad \text{(inspb-dsc-eou)}$$

Figure 15: Semantics for filtered iterators—with bad binary inputs

$$\frac{\text{file } str \text{ does not exist}}{\langle f = \texttt{opent}(str),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma,\mathrm{bad}\rangle} \quad \text{(opent-bad)} \qquad \frac{\text{file } str \text{ does not exist}}{\langle f = \texttt{openb}(str),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma,\mathrm{bad}\rangle} \quad \text{(openb-bad)}$$

$$\frac{\text{file } str \text{ exists}}{\langle f = \texttt{opent}(str),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \quad \langle\langle \sigma_S[f\mapsto \mathrm{hndl}],\sigma_H,\sigma_F[\mathrm{hndl}\mapsto\langle str,0,0,\langle\varnothing,\varnothing,\varnothing\rangle\rangle],\sigma_D\rangle,\mathrm{ok}\rangle} \quad \text{(opent-ok)}$$

$$\frac{\text{file } str \text{ exists} \quad \sigma_D(str) = \langle \gamma,n\rangle}{\langle f = \texttt{openb}(str),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \quad \langle\langle \sigma_S[f\mapsto \mathrm{hndl}],\sigma_H,\sigma_F[\mathrm{hndl}\mapsto\langle str,0,0,n\rangle],\sigma_D\rangle,\mathrm{ok}\rangle} \quad \text{(openb-ok)}$$

Figure 17: Semantics for opening input files

$$\frac{\langle e,\sigma\rangle \Downarrow_{\mathrm{e}} \mathrm{err}}{\langle \texttt{seek}(f,e),\sigma\rangle \Downarrow_{\mathrm{s}} \langle \sigma,\mathrm{bad}\rangle} \quad \text{(sk-bad)}$$

$$\frac{\langle e,\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_F(\sigma_S(f)) = \langle str,l,sou,ud\rangle \quad u < sou}{\langle \texttt{seek}(f,e),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle,\mathrm{bad}\rangle} \quad \text{(sk-l)}$$

$$\frac{\langle e,\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_F(\sigma_S(f)) = \langle str,l,sou,ud\rangle \quad \sigma_D(str) = \langle \gamma,n\rangle \quad sou \leq u \leq \Lambda(l)}{\langle \texttt{seek}(f,e),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \quad \langle\langle \sigma_S,\sigma_H,\sigma_F[\sigma_S(f)\mapsto\langle str,u,sou,ud\rangle],\sigma_D\rangle,\mathrm{ok}\rangle} \quad \text{(sk-ok)}$$

$$\frac{\langle e,\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{e}} u \quad \sigma_F(\sigma_S(f)) = \langle str,l,sou,ud\rangle \quad \sigma_D(str) = \langle \gamma,n\rangle \quad u > \Lambda(l)}{\langle \texttt{seek}(f,e),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle,\mathrm{bad}\rangle} \quad \text{(sk-r)}$$

$$\frac{\sigma_F(\sigma_S(f)) = \langle str,l,sou,ud\rangle \quad \sigma_D(str) = \langle \gamma,n\rangle \quad l = \Lambda(l)}{\langle x = \texttt{read}(f),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle,\mathrm{bad}\rangle} \quad \text{(frd-out)}$$

$$\frac{\sigma_F(\sigma_S(f)) = \langle str,l,sou,ud\rangle \quad \sigma_D(str) = \langle \gamma,n\rangle \quad l < \Lambda(l)}{\langle x = \texttt{read}(f),\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_{\mathrm{s}} \quad \langle\langle \sigma_S[x\mapsto\gamma(l)],\sigma_H,\sigma_F[\sigma_S(f)\mapsto\langle str,l+1,ud\rangle],\sigma_D\rangle,\mathrm{ok}\rangle} \quad \text{(frd-ok)}$$

Figure 18: Semantics for accessing input files

MIT and included both doctoral students and post-doctoral researchers at MIT.

We manually matched pairs of subjects based on educational background, programming experience in the past five years, and knowledge of C/C++. We randomly assigned the two subjects in each pair to either the inspect group or the control group. Both groups performed the programming task using a virtual machine that we prepared for this experiment. We then used the Wilcoxon signed-rank test [77], which tests the location shift of paired samples, to analyze the resulting developer programs.

Independent Variable: The only difference between the two groups was the use of the `inspect` construct: the inspect group used the full RIFL language, while the control group used RIFL without the `inspect` construct, but with standard control flow constructs and error signaling.

Dependent Variables: We collected and analyzed all programs in terms of defects, behavior on a range of inputs, and complexity. We are most interested in the number of total defects.

- **Defects:** To address **RQ1**, we manually analyzed the defects in all programs from the experiment, based on a predefined list of possible defects which we extended during the analysis. Tables 2 and 3 present the lists of possible fatal defects and other defects, respectively, where the first columns present abbreviations and the second columns describe the defects in detail. We decided prior to the experiment to count each defect once for each program.
- **Behavior on test inputs:** To address **RQ2** and **RQ3**, we designed a range of test inputs to expose possible defects. For each test input, we compared the behavior of the two programs from each pair of subjects. These test inputs include *common legal* inputs that test the main functionality of the programs, *rare legal* inputs that still satisfy the input specification but contain corner cases that developers may not handle correctly, and *illegal* inputs that are designed to test the error detection and handling code. Similar to the list of defects, we started with a predefined list of inputs, and included more to trigger new defects that we observed from code review. Appendix A presents all these inputs and their correct outputs.
- **Comparison with failure-oblivious computing:** To address **RQ4**, we implemented a failure-oblivious version of the interpreter for the control group's language. This interpreter has the same behavior with the original control group interpreter in benign situations and differs in erroneous situations that would originally cause crashes. Specifically, the failure-oblivious interpreter (a) returns value zero for divide-by-zero expressions, out-of-bound array reads, and null array reads and (b) silently ignores out-of-bound array writes and null array writes.

  For each test input above, we observed the behavior of the control group programs running on the failure-oblivious interpreter. For reference, we also analyzed the potential behavior of programs that entered infinite loops if there are tools [14] that help programs (a) escape from infinite loops and (b) continue executing the instructions that follow the escaped loop.
- **Program complexity:** To address **RQ5**, we measured the complexity of the programs in terms of control flow, data manipulation, and total lines of code. These measurements indicate the difficulty of program implementations.
  - **Control-flow complexity:** To estimate the difficulty of error detection, we observe the number of *conditional clauses* in programs. This number contains `if` statements, `assert` statements, logical conjunctions, and logical disjunctions.
  - **Data-manipulation complexity:** To estimate the difficulty of error recovery, we observe the number of *unconditional statements* in programs. This number contains all statements except for `if` and `assert`.
  - **Lines of code:** For reference, we also observe the numbers of lines in the source file, including lines that contain no statements.

Experimental Procedure: The experiment contains a language tutorial and the thumbnail generator task. In the tutorial stage, each subject reads a language manual, runs an

Table 2: Possible fatal defects. New defects observed are underlined.

Defect	Description
AWL	Out-of-bound array write when reading input, triggered by input fields that are longer than an input buffer.
AWO	Out-of-bound array write when reading input, triggered by an integer overflow that causes overly small memory allocation.
ARL	Out-of-bound array read during computation, triggered by image dimensions that are too large for an input buffer.
ARO	Out-of-bound array read during computation, triggered by an integer overflow that causes overly small memory allocation.
DS	Division by zero during computation, triggered by a zero scaling factor.
DD	Division by zero when checking integer overflow, triggered by a zero dimension.
NA	Null array access when reading input, triggered by failed memory allocation.
IL	Infinite loop when reading illegal input units.

Table 3: Possible other defects. New defects observed are underlined.

Defect	Description
MP	Memory leak even when processing common legal input units.
MS	Memory leak when skipping input units.
WP	Wrong behavior from producing partial outputs for illegal input units.
WS	Wrong behavior from de-synchronization for at least one input unit after illegal input units.
WM	Wrong behavior from misusing illegal input units and producing outputs for these illegal input units as if they are legal.
WA	Wrong behavior from aborting on illegal input units.

example program, and writes two small programs that may be helpful for implementing the thumbnail generator. These two small writing tasks are important because (a) they interactively *teach* subjects to write and debug with in the experimental language, (b) they *calibrate* two groups to ensure that all subjects from both groups understand the experimental language to similar levels after the tutorials, and (c) they allow us to *detect outliers* based on the subjects' abilities to write and learn the experimental language.

The specification for the thumbnail generator task explains the program functionality with typical inputs and states that the program should be able to handle arbitrary inputs. For reference, the complete paper materials are presented in Appendix B. The experiments are not limited in duration.

To gain more insight into the developers' experience, we recorded full screen recordings of the developers as they worked and included a post-experiment questionnaire. The questionnaire asked the subjects for feedback by (a) asking the inspect group to rate the helpfulness of the `inspect` construct, (b) asking all subjects to rate the level of their perceived difficulty [38], and (c) asking for additional comments or clarification.

4.2 Results

Power Analysis: Prior to collecting data, we first estimated the number of subjects we would need to observe statistically significant results. We performed a power analysis [30] using the statistical package R [54]. The power analysis showed that five pairs of subjects would give us 41% statistical power ($1 - \beta = 0.41$)[2] at a standard 0.05 level ($\alpha = 0.05$), given that we are interested only in large effect sizes [16, 30]. We then performed a pilot experiment to confirm that the effect size for the number of defects–our most important measurement–was likely to be large.

Subjects: Twelve subjects participated. None of them had any prior experience with RIFL. None of them knew about the design or the goals of the study except that it was about language features and program complexity. All subjects volunteered for the study for five dollars' compensation.

We name each subject with a letter that stands for the group and a number that stands for the pair. Letters "i" and "e" denote the inspect group and the control group, respectively. Among all subjects, ten were matched up into five pairs (1, 2, 4, 5, and 6), based on the similarity of their backgrounds.

Two subjects (e3 and i7) were outliers, because their programs do not produce correct outputs even for common legal inputs, there are no straightforward ways to correct the defects, and the subjects did not learn from solutions to the tutorial problems. Consequently, we removed these outliers from the statistical analysis and did not perform their counterpart experiments (there were no i3 or e7).

[2] This power is the probability for us to detect statistical significance when there is true difference between the two groups.

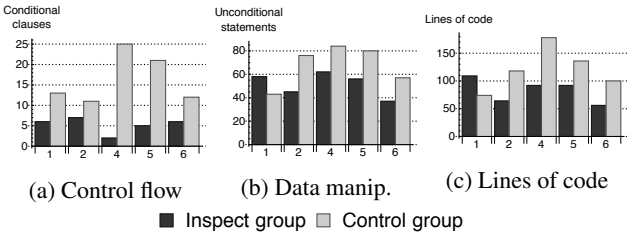


Figure 19: Complexity measurements for user study

We observed that six subjects did not check that their programs generated outputs that matched the specification precisely. Trivial fixes, such as adding the value '`0`' to the final result, brought the programs into compliance. Two of these fixes (i5 and e6) were applied immediately on the scene after the participants said that they had finished, when the problem was brought to their attention. The other four (i1, e1, e2, and i4) were fixed during our analysis.

Most subjects took comparable time, between one and two hours, to finish the entire experiment. Three subjects (e1, e2, and i7) took between two and three hours. One subject (e3) took more than three hours.

Measurements: Table 4 presents the defects that we found in all the developer programs. Each column represents a thumbnail program by the name of the developer. The rows are: the number of fatal defects ("# fatal"), the list of fatal defects (next eight rows), the number of other defects ("# other"), the list of other defects (next six rows), and the number of all defects ("Total").

Table 5 presents the behavior of these programs processing the test inputs. Each column represents a thumbnail program. We denote these failure-obliviously executed control group programs with "foc" followed by the pair number. The rows are: the behavior on common legal inputs (the first row), the number of crashes or infinite loops on rare legal inputs ("# crash/loop rare"), the number of wrong outputs on rare legal inputs ("# wrong/explode rare"), the behavior on rare legal inputs (next four rows), the number of crashes or infinite loops on illegal inputs ("# crash/loop illegal"), the number of wrong outputs on illegal inputs ("# wrong/explode illegal"), the behavior on illegal inputs (next 21 rows), the number of crashes or infinite loops on all inputs ("# crash/loop total"), and the number of wrong outputs on all inputs ("# wrong/explode illegal"). We define a wrong output to *explode* if the output size is proportional to the input value, instead of to the input size.

Figure 19 presents the complexity measurements for all the developer programs. The vertical axes in Figures 19a, 19b, and 19c represent control complexity, data complexity, and lines of code, respectively. Each group of two bars represents the two programs from a pair of subjects.

Statistical Analysis: We applied the one-sided Wilcoxon signed-rank tests [77] on all measurements using R [54]. We

Table 5: Behavior of the inspect group programs, original control group programs, and failure-obliviously executed control group programs processing the test inputs. Letters "C", "L", "W", and "E" denote crashing, entering an infinite loop, producing wrong and small outputs, and producing wrong and exploding outputs, respectively. Letter "s" denotes data de-synchronization for at least one subsequent input unit. Letter "a" denotes aborting. The combination "Ls" denotes that the program enters an infinite loop and that the program would de-synchronize after escaping from the infinite loop. Empty cells denote correct behavior. The inspect group is underlined.

Input	Category	i1	e1	foc1	i2	e2	foc2	i4	e4	foc4	i5	e5	foc5	i6	e6	foc6
good	common															
# crash/loop rare		**0**	**1**	**0**	**0**	**2**	**0**	**0**	**1**	**0**	**0**	**2**	**0**	**0**	**1**	**0**
# wrong/explode rare		**0**	**0**	**1**	**0**	**0**	**2**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**1**
bufovfverylongname	rare					C	W					C				
div0h	rare															
div0w	rare															
heapovf2	rare		C	W		C	W		C	W		C			C	W
# crash/loop illegal		**0**	**10**	**0**	**0**	**12**	**4**	**0**	**2**	**0**	**0**	**7**	**0**	**0**	**6**	**0**
# wrong/explode illegal		**0**	**7**	**16**	**0**	**7**	**14**	**2**	**0**	**1**	**1**	**3**	**8**	**4**	**11**	**17**
bufovfint1	illegal		C	E		C	E					C	E		C	Es
bufovfint2	illegal		C	E		C	E					C	E		C	Es
bufovfint3	illegal		C	E		C			C	E		C	E		C	E
bufovfint4	illegal		C	E		C	W					C	E		C	Es
charh	illegal		W	W		Ws	Ws					W	W	W		
charpix1	illegal		W	W		Ws	Ws	W			W			W	Wa	Wa
charpix2	illegal					Ls	Ls								Wa	Wa
chars	illegal		W	W		C	W	W							W	W
chartrail	illegal					Ws	Ws								Ws	Ws
charw1	illegal		W	W		Ws	Ws					W	W	W		
charw2	illegal		C	W		W	W								Ws	Ws
div0s	illegal		C	W		C	W					C	W		C	W
empty	illegal															
heapovf1	illegal		C	E		C	Ls		C			C			C	E
intovf	illegal															
long	illegal		C			C	Ws					C			Ws	Ws
nullint	illegal		W	W		Ws	Ws					Wa	Wa	W	Ws	Ws
short1	illegal		W	W		Ls	Ls								W	W
short2	illegal		C	W		Ws	Ws								Ws	Ws
short3	illegal		W	W		Ls	Ls								W	W
short4	illegal		C	W		C	Ws								Ws	Ws
# crash/loop total		**0**	**11**	**0**	**0**	**14**	**4**	**0**	**3**	**0**	**0**	**9**	**0**	**0**	**7**	**0**
# wrong/explode total		**0**	**7**	**17**	**0**	**7**	**16**	**2**	**0**	**2**	**1**	**3**	**8**	**4**	**11**	**18**

assume that the differences are comparable across pairs [69]. We observed statistical significance to reject null hypotheses where the p-values are less than the standard significance level, 0.05.

- **Defects, RQ1:** According to the results of the tests, the inspect group has significantly fewer defects (p-value = 0.029) and fewer fatal defects (p-value = 0.029) than the control group. The inspect group also tends to have fewer non-fatal defects than the control group (p-value = 0.068).
- **Survivals, RQ2:** The inspect group crashes or enters infinite loops on significantly fewer occasions than the control group when processing rare legal inputs (p-value = 0.027), illegal inputs (p-value = 0.031), and all inputs (p-value = 0.031).
- **Wrong outputs, RQ3:** The inspect group tends to produce fewer wrong outputs than the original control group when processing illegal inputs (p-value = 0.064).
- **Wrong outputs with failure-oblivious computing, RQ4:** The inspect group produces fewer wrong outputs than the failure-obliviously executed control group programs when processing rare inputs (p-value = 0.044). The inspect group tends to produce fewer wrong outputs than the failure-oblivious control group when processing illegal inputs (p-value = 0.063) and all inputs (p-value = 0.050).
- **Program complexity, RQ5:** The inspect group programs have significantly fewer conditional clauses (p-value = 0.031) than the control group programs do. The inspect group programs also tend to have fewer unconditional statements (p-value = 0.063) and fewer lines of code (p-value = 0.052).

Our overall conclusion is that these results support the hypothesis that the RIFL filtered iterators can enable a range of developers to produce safe and robust programs with significantly fewer defects than comparable developers using standard language constructs. We also note that the numerous defects—including defects corresponding to typical secu-

Table 4: Defects in developer programs. Letter "X" denotes the existence of a defect. The inspect group is underlined.

Defect	i1	e1	i2	e2	i4	e4	i5	e5	i6	e6
# fatal	**0**	**4**	**0**	**5**	**0**	**2**	**0**	**4**	**0**	**3**
AWL		X		X				X		
AWO										
ARL		X								
ARO				X		X		X		X
DS		X		X				X		X
DD										
NA		X		X		X		X		X
IL				X						
# other	**1**	**2**	**0**	**4**	**1**	**0**	**2**	**4**	**2**	**5**
MP		X					X	X	X	X
MS	X			X				X		
WP				X						X
WS				X						X
WM		X		X	X		X	X	X	X
WA								X		X
Total	**1**	**6**	**0**	**9**	**1**	**2**	**2**	**8**	**2**	**8**

rity vulnerabilities—present in the control group programs are consistent with the defects observed in input-processing code more broadly. Filtered iterators eliminate the vast majority of these defects.

4.3 Discussion

We attribute the statistically significant results from only five pairs of subjects to the fact that the difference between the inspect group and the control group is very large.

Program Robustness: Defects, survivals, and wrong outputs are three measures of program robustness. RIFL's filtered iterators enhance input-parsing programs' robustness by reducing defects, increasing survivals, and reducing wrong outputs. We attribute these improvements to the fact that filtered iterators automatically filter out bad input units and allow programs to continue execution from an earlier, consistent state.

Program Complexity: RIFL's filtered iterators reduce the complexity of input-parsing programs by significantly reducing the number of conditional clauses. For the other two aspects—unconditional statements and lines of code—the complexity measurements were reduced somewhat, though not statistically significant[3]. We attribute these improvements to the fact that the semantics of filtered iterators ensure the atomicity of input units.

Participants' Feedback: Two subjects from the inspect group (i1 and i2) responded that the `inspect` construct was very helpful for implementation. One of them (i2) reported verbally that `inspect` frees him/her of the needs to consider many corner cases. The other (i1) reported verbally that, without a good integrated development environment (IDE)[4], he/she had to debug manually by inserting `print` and `continue` statements. Two subjects (i4 and i5) responded that `inspect` was somewhat helpful, but also makes debugging hard without a good IDE, because it makes some unintentional errors silent. One subject (i7) responded that `inspect` was somewhat useless during development, because he/she would like his/her programs to "blow up and scream at" him/her when something is wrong. The last inspect group subject (i6) responded that `inspect` was useless, and did not clarify further on this point. Despite this comment, we note that, using the `inspect` construct, this subject produced a program with significantly fewer errors than his/her counterpart (e6).

This developer feedback highlights a difference between software that is under development and software that is in production. If deployed during development, mechanisms that help software tolerate errors may inhibit the development of quality software by triggering a bystander effect [21, 42, 57] that demotivates developers to improve the software. Understanding how this phenomenon could affect RIFL developers requires some sophistication. One of the goals of RIFL is to free developers from the need to produce code that handles corner-case input units that the application should discard. Ideally, RIFL developers would not even have to mentally conceptualize or consider such cases. In this situation, the RIFL mechanisms constitute a beneficial bystander that enhances developer productivity by reducing the developer's mental load.

The potential drawback comes when the developer makes unintentional errors when processing input units that the application should handle properly. One way to navigate this dichotomy of inputs is to develop software with a more sophisticated testing approach—classifying test inputs into (a) tests that the application should fully process and (b) tests containing input units that the application should discard. This categorization enables RIFL developers to highlight the inputs that the application should fully process by configuring the RIFL runtime to raise visible errors for these inputs, instead of silently discarding input units that trigger errors. This enhancement would properly motivate developers to produce code that correctly implements the basic functionality of the application. This approach would both (a) preserve the benefits of the RIFL mechanisms that implicitly discard input units that the program cannot correctly process and (b) provide the visible error feedback that helps developers find and eliminate errors for which they are responsible. We note that there is a conceptual similarity to just in time manufacturing approaches, which are designed to drive the development of effective manufacturing processes by making errors immediately visible.

[3] From our manual examination of the developers' programs, we attribute this lack of significance to the noise caused by various approaches to implementing the same functionality.

[4] Unlike the complete RIFL interpreter, the interpreters developed for the controlled experiment produced only final outputs for both groups. There were no error logs that could help developers debug programs when inspect loops discard bad input units.

5. Other Input File Formats

We also investigated the use of RIFL for the following input file formats. Binary formats include PCAP/DNS, RGIF, PNG, and ZIP. Text formats include JSON, OBJ, and CSV. More detail about these benchmarks, including source code, are presented in Appendix C.

- **PCAP/DNS:** A network traffic capture [6] format, PCAP [23], stores network packets. The PCAP/DNS benchmark implementations analyze Domain Name System (DNS) packets from PCAP files, printing the Internet Protocol (IP) addresses, the DNS identification number, and the DNS questions and answers.
- **RGIF:** We adapt the Graphics Interchange Format (GIF) [3], which stores animated images, to become more resilient to malformed image components. We call this new format RGIF. It differs from GIF by adding two bytes in front of each image to describe the length of all blocks for the image. The RGIF benchmark implementations convert an RGIF file to the GIF format.
- **Portable Network Graphics (PNG):** The PNG format [11] stores images. The PNG benchmark implementations concatenate all image data in an PNG file.
- **ZIP:** The ZIP format [52] stores archived user files. The ZIP benchmark implementations output the name and the archived contents for all user files.
- **JavaScript Object Notation (JSON):** The JSON format [12] describes an object. The JSON benchmark implementations copy inputs to outputs, discarding any bad components.
- **OBJ:** OBJ [4] files store three-dimensional (3D) object geometry. The OBJ benchmark implementations copy inputs to outputs, discarding any bad components.
- **Comma Separated Values (CSV):** CSV [63] files store tables. The CSV benchmark implementations copy inputs to outputs, discarding bad components.

5.1 Methodology

We implemented four versions of each program: (a) *Full RIFL:* this version uses inspect loops to automatically extract input units, to detect bad units, and to recover from bad units, (b) *Explicit Check:* this version uses inspect loops with exhaustive assertions to explicitly identify bad input units that the RIFL implementation automatically discards on these assertion failures, (c) *Explicit Recovery:* this version uses RIFL iterators but with fully explicit error handling so the RIFL error detection and recovery mechanisms never deploy, and (d) *Conventional:* this version uses conventional language constructs.

All versions of the same benchmark program have the same functionality and produce the same outputs for most inputs. In practice, it is sometimes inappropriate to require exactly the same outputs. For example, when a program rejects an input file entirely because the input significantly differs from the specification, the full-RIFL and explicit-check versions may use assertions outside inspect loops, while it may be more natural for the other two versions to use `exit` statements that terminate programs with error codes and partial outputs.

We compared all versions of each program in terms of control complexity, data complexity, and lines of code. We also analyzed the differences between the source code for all versions of the same programs and classified the differences into importance categories.

5.2 Results

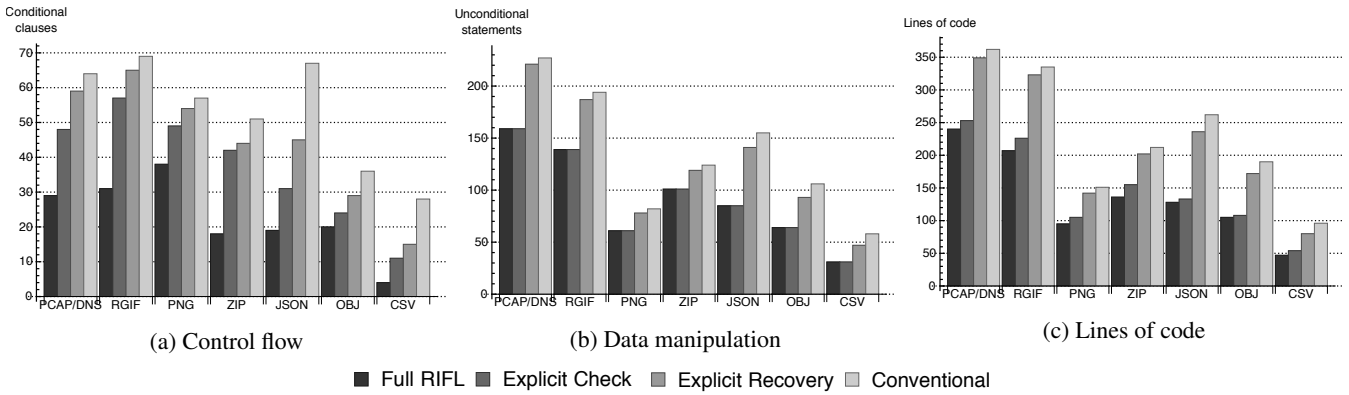
Figure 20 presents the measurements for all versions of all benchmark programs. The vertical axes in Figures 20a, 20b, and 20c represent control complexity, data complexity, and lines of code, respectively. Each group of four bars represents the four versions of a program.

Tables 6, 7, and 8 present the average percentage decrease in control complexity, data complexity, and lines of code from each version. The columns are averages for binary formats (Binary), averages for text formats (Text), and averages for all seven benchmarks (Overall). The rows are average percentage decreases in complexity: from conventional to explicit-recovery versions (Iterator), from explicit-recovery to explicit-check versions (Implicit recovery), from explicit-check to full-RIFL versions (Implicit detection), and from conventional to full-RIFL versions (Overall). Filtered iterators reduce the numbers of conditional clauses and unconditional statements from conventional versions by an average of 58.5% and 33.4%, respectively. Overall, filtered iterators reduce the number of lines of code by an average of 41.7%.

Table 9 presents our perception of the categories of the error-handling code that the full-RIFL versions eliminate from the conventional versions. Each row corresponds to a benchmark program. Columns two through four correspond to importance categories: preventing crashes (Vital), preventing input misinterpretation (Integrity), and part of common engineering practice (Common). The last two columns correspond to the total numbers eliminated in all three categories (Total) and the total numbers present in the conventional versions (Size), respectively.

5.3 Discussion

We attribute the improvements of eliminated error-handling code to the fact that the full-RIFL versions need only code that handles common cases. Specifically: (a) Iterators can help eliminate the code that identifies the boundaries between input units and prevents de-synchronization in case of bad input units. (b) Implicit error recovery can help eliminate the code that maintains program state and avoids partial outputs when discarding bad input units. (c) Implicit error detection can help eliminate the conditionals that prevent program crashes by checking corner cases that developers may not consider. The omission of these conditionals in con-



(a) Control flow (b) Data manipulation (c) Lines of code

Figure 20: Complexity measurements for benchmarks

Table 6: Average decrease in control flow

Feature	Binary	Text	Overall
Iterator	8.1%	32.9%	18.8%
Implicit recovery	11.2%	25.0%	17.1%
Implicit detection	41.2%	39.7%	40.5%
Overall	51.9%	67.3%	**58.5%**

Table 7: Average decrease in data manipulation

Feature	Binary	Text	Overall
Iterator	3.8%	13.4%	7.9%
Implicit recovery	22.7%	35.0%	27.9%
Implicit detection	0.0%	0.0%	0.0%
Overall	25.6%	43.8%	**33.4%**

Table 8: Average decrease in lines of code

Feature	Binary	Text	Overall
Iterator	4.5%	12.0%	7.7%
Implicit recovery	26.7%	37.8%	31.5%
Implicit detection	8.8%	6.5%	7.8%
Overall	36.2%	49.0%	**41.7%**

Table 9: Error-handling code totals for benchmarks. Each pair of numbers separated by a slash "/" are the numbers of conditional clauses and of unconditional statements.

Benchmark	Vital	Integrity	Common	Total	Size
PCAP/DNS	8 / 9	26 / 58	1 / 1	35 / 68	64 / 227
RGIF	10 / 5	27 / 50	1 / 0	38 / 55	69 / 194
PNG	4 / 3	13 / 18	2 / 0	19 / 21	57 / 82
ZIP	7 / 1	16 / 19	10 / 3	33 / 23	51 / 124
JSON	13 / 4	32 / 62	3 / 4	48 / 70	67 / 155
OBJ	5 / 3	10 / 38	1 / 1	16 / 42	36 / 106
CSV	4 / 1	18 / 25	2 / 1	24 / 27	28 / 58

ventional versions would, therefore, typically lead to errors when processing the inputs with these corner cases.

6. Related Work

The notion of filtered iterators combines filters, iterators and atomic transactions with input units. Filtered iterators treat inputs as collections of input units, iterate over the units, use the programs to filter out bad units, and atomically update program state for each unit. A filter function extracts a subset from a collection so that each element satisfies a given predicate. Filters date back to the `remove-if` function in Common Lisp [70] and the `select` message in Smalltalk [32]. An iterator is a generalization of loops over collections, which separates the action performed on each object from the selection of the objects. The concept of iterators was originally proposed in CLU [44] as a control abstraction. RIFL abstracts the input as a structured collection of input units. Many modern languages [2, 5] support iterators with explicit filtering predicates. RIFL, in contrast, uses the execution of the program itself as an implicit filter. The result is automatic detection and recovery from a range of errors. Loop perforation [67] drops loop iterations from the computation. RIFL drops input units from the inputs. A transaction is a group of actions with ACID properties: atomicity, consistency, isolation and durability. Transactions ensure the consistency in spite of concurrency and failures for database systems [31, 35, 36] and for distributed systems [43]. Transactions are used as an alternative abstraction to explicit synchronization [45]: transactional memory [39, 64, 71] implementations simplify the management of concurrent data structures. RIFL uses transactions in filtered iterators to enforce the atomicity of input units.

Researchers have proposed ways to recover software from errors. Many of them are external to the language definition. Transactional recovery techniques such as backward recovery [15] and forward recovery [40] recover programs from transient errors. Rx [53] rolls back and replays programs in new configurations to survive failures. These techniques are unlikely to solve deterministic errors. Filtered

iterators capture all detectable errors and recover from them by discarding bad inputs and restarting loop iterations.

Input filtering systems [13, 17, 18, 47, 73, 75] generate vulnerability signatures from known attacks and drop malicious inputs accordingly. These techniques are usually unsound—they cannot block all attacks. Input rectification [46, 58] prevents program failures by modifying bad inputs to good ones using prior knowledge about benign inputs. The RIFL runtime dynamically captures and recovers from all detectable vulnerabilities with neither false positives nor false negatives. RIFL also uses programs themselves as filters, without external knowledge.

Failure-oblivious techniques [48, 57] purposefully change the program semantics to survive inputs that the program would otherwise be unable to process. Error virtualization [65, 66] recovers program execution by turning function executions into transactions and mapping faults into return values used by the application code. These techniques aim to change the semantics of existing programs to improve robustness, but with at best uncertain impacts on the semantics of the program. RIFL, in contrast, provides semantics that have no anomalous effects from bad inputs.

Researchers have also proposed language-based techniques to error recovery. Recovery blocks [7] and N-version programming [8] tolerate software faults using multiple implementations of the same component. Exception handling [19, 33] improves program structures by separating common-case and error-handling code. This structure requires developers to anticipate the exceptional events and maintain program state accordingly. In contrast to these methodologies, RIFL aims at eliminating the need for error-handling code. Bristlecone [24, 26, 27] ensures error-free execution using decoupled transactional tasks according to high-level task specifications. RIFL and Bristlecone both help programs survive by discarding computation from the conceptual level. RIFL's filtered iterators allow program structures to be closer to existing programs than Bristlecone's task descriptions would.

7. Conclusion

Input validation defects are a common and serious source of software errors and security vulnerabilities. By dividing inputs into input units and atomically discarding input units that trigger errors, RIFL promotes the development of reliable and robust software systems. Statistically significant results from a developer study demonstrate the empirical benefits that RIFL can provide.

More generally, RIFL embodies a different approach to programming language design. Instead of operating from abstract philosophical principles such as simplicity, orthogonality, or elegance, RIFL targets a specific problem that has emerged through experience with existing languages over time, provides a construct designed to emulate the effect of observed developer fixes, and validates its approach using empirical techniques rather than abstract philosophical arguments. As the field matures and acquires the experience required to more accurately identify the important issues that actually occur in practice, we expect this more empirical approach to comprise one effective approach to programming language design.

A. Inputs for Testing Programs from the Developer Study

A.1 Common Legal Inputs

- **Input "good":**

```
Img1␣2␣2␣2␣1234
Img2␣2␣4␣4␣1234567890123456
Img3␣2␣1␣2␣12
Img4␣3␣3␣4␣123456789012
Img5␣5␣5␣10␣
     0123456789012345678901234567890123456789012345678901234567890123456789
```

Correct output:

```
Img1␣2
Img2␣3543
Img3␣
Img4␣3
Img5␣27
```

A.2 Rare Legal Inputs

- **Input "bufovfverylongname":**

```
BufOvfVeryLongName␣2␣2␣2␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
BufOvfVeryLongName␣2
Img5␣2
```

- **Input "div0h":**

```
Div0H␣2␣0␣2
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
Div0H␣
Img5␣2
```

- **Input "div0w":**

```
Div0W␣2␣2␣0
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
Div0W␣
Img5␣2
```

- **Input "heapovf2":** The first line contains "`HeapOvf2␣1␣200␣1000␣`" followed by 20000 repetitions of "`1234567890`". The second line contains "`Img5␣2␣2␣2␣1234`". Correct output:

```
Img5␣2
```

or
The first line contains "`HeapOvf2␣`" followed by 20000 repetitions of "`1234567890`". The second line contains "`Img5␣2`".

A.3 Illegal Inputs

- **Input "bufovfint1":**

```
BufOvfInt1␣2␣16␣268435457␣123456789012345678901234567890
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input "bufovfint2":**

```
BufOvfInt2␣2␣268435457␣16␣123456789012345678901234567890
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input "bufovfint3":**

```
BufOvfInt3␣1␣268435457␣16␣1234567890123456
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
BufOvfInt3␣1234567890123456
Img5␣2
```

- **Input "bufovfint4":**

```
BufOvfInt4␣1␣268435457␣16␣123456789012345678901234567890
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input "charh":**

```
CharH␣2␣␣2␣2␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
CharH␣2
Img5␣2
```

- **Input "charpix1":**

```
CharPix1␣2␣2␣2␣12a4
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input "charpix2":**

```
CharPix2␣3␣3␣4␣123a567b901c
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
CharPix2␣3
Img5␣2
```

- **Input "chars":**

```
CharS␣b␣2␣2␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input "chartrail":**

```
CharTrail␣2␣2␣2␣1234b
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
CharTrail␣2
Img5␣2
```

- **Input "charw1":**

```
CharW1␣2␣2␣␣2␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
CharW1␣2
Img5␣2
```

- **Input "charw2":**

```
CharW2␣2␣2a␣2␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input "div0s":**

```
Div0S␣0␣2␣2␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input “empty”:**

```
␣2␣2␣2␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
␣2
Img5␣2
```

- **Input “heapovf1”:**

```
HeapOvf1␣2␣200␣1000␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input “intovf”:**

```
IntOvf␣2␣2␣4294967298␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

or

```
IntOvf␣2
Img5␣2
```

- **Input “long”:** The first line contains “`Long␣2␣2␣2␣`” followed by eleven repetitions of “`1234567890`”. The second line contains “`Img5␣2␣2␣2␣1234`”. Correct output:

```
Img5␣2
```

or

```
Long␣2
Img5␣2
```

- **Input “nullint”:**

```
NullInt␣2␣2␣4294967295␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input “short1”:**

```
Short1␣2
2␣2␣1234
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input “short2”:**

```
Short2␣2␣2␣2␣
123
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input “short3”:**

```
Short3␣2␣4␣4␣
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

- **Input “short4”:**

```
Short4
Img5␣2␣2␣2␣1234
```

Correct output:

```
Img5␣2
```

B. Paper Materials for the Developer Study

B.1 Consent Form

CONSENT TO PARTICIPATE IN EXPERIMENT

Title of the Study: Language Features and Program Complexity

You have been asked to participate in a research study conducted by Jiasi Shen from the Department of Electrical Engineering and Computer Science (EECS) at the Massachusetts Institute of Technology (M.I.T.). The purpose of the study is to investigate how the features of a programming language affect the difficulty of reading and writing programs. The results of this study will be published as a report. You were selected as a possible participant in this study because you are an experienced software developer. You should read the information below, and ask questions about anything you do not understand, before deciding whether or not to participate.

• This experiment is voluntary. You have the right not to answer any question, and to stop the experiment at any time or for any reason. We expect that the experiment will take 30-60 minutes.

• You will be compensated $5 for this experiment.

• Your name will remain confidential. Unless you give us permission to quote you in any publications that may result from this research, the information you tell us will be confidential.

• You will be asked to read and write programs inside a virtual environment. For analyzing purposes, we will record the *visual* information that this virtual environment displays as you perform the tasks. The information we learn from these recordings will be made anonymous in our reports.

This project will be completed by July 2016.

(*Please check all that apply*)

[ ] I understand the procedures described above. My questions have been answered to my satisfaction, and I agree to participate in this study. I have been given a copy of this form.

[ ] I give permission for direct quotes from this experiment to be included in publications resulting from this study.

Name of Subject

Signature of Subject _____ Date _____

Signature of Investigator _____Date _____

Please contact Jiasi Shen (jiasi@csail.mit.edu) with any questions or concerns.

If you feel you have been treated unfairly, or you have questions regarding your rights as a research subject, you may contact the Chairman of the Committee on the Use of Humans as Experimental Subjects, M.I.T., Room E25-143b, 77 Massachusetts Ave, Cambridge, MA 02139, phone 1-617-253-6787.

Amendment Approved on 03-NOV-2015

Exemption Granted 17-AUG-2015

B.2 Inspect Group Instructions

Instructions

You are going write three programs using a new programming language. Please follow the instructions below to confirm your eligibility, setup the virtual machine, learn the language, complete the writing tasks, complete surveys regarding your experience, and submit your data.

1 Eligibility

In the **past five years**, you have _____ years' experience using imperative languages _____ (such as C/C++, Java, or Python).

2 Environmental setup

1. Launch the virtual machine by opening the file "LanguageFeaturesVM.vmwarevm" with VMware Fusion.
2. Adjust the VMware window to your screen, so that you can see both the lower-left corner with a trash-can icon and the upper-right corner with a gadget icon.
3. Launch the screen recorder by clicking the blue monitor icon on the left. A window titled "vokoscreen 1.9.0" should appear. [1]
4. Click "Start" in the vokoscreen window. The vokoscreen window should minimize automatically, and there should be a small white triangle to the left of the blue monitor icon that you have previously clicked.
5. Launch a terminal by clicking the black rectangle icon on the left. A terminal window titled "default@ubuntu:~\Desktop" should appear.

[1] For analyzing purposes, we record the screen in the virtual machine as you perform tasks in this experiment. We have configured vokoscreen to record the full screen, without audio, capturing 25 frames per second, encoding in the x264 format, storing into an avi file, and including mouse cursor movements. Feel free to double check these configurations and ask the experimenter if you have any questions.

3 Language tutorial

The language used in this experiment is similar to many other imperative languages such as C, but includes an additional looping construct: `inspect`.

1. Learn about `inspect` loops by reading *The `inspect` Construct*.
2. Learn about the rest of the language by reading the *Language Manual*.
3. **Double-click** the file "prog0.stu" on the desktop. An editor window titled "prog0.stu (~\Desktop) – gedit" should appear. This file contains an example program that counts the number of characters on each line of the file "input0.txt".
4. Run this example program by entering

   ```
   run prog0.stu
   ```

   in the terminal. You should see

   ```
   [Program return value]: 0.
   \5
   \3\0
   ```

   in the same terminal window. The first line presents the value returned from the program's `main` function. Then follows the outputs from `print` statements. The strings "\5", "\3", and "\0" indicate that there are 5, 3, and 0 characters on the first, the third, and the fourth lines, respectively. The newline character between "\5" and "\3" indicates that there are 10 characters on the second line, since the number 10 corresponds to the newline character `'\n'` in ASCII encoding.
5. Feel free to edit files "prog0.stu" and "input0.txt" and run the program again.

4 Writing tasks

There are three writing tasks. The first two tasks should help you complete the third task. Please complete them in order, and fill in the time that you start and finish each task.

4.1 Summing Matrix Elements

Started reading at _____ : _____ : _____

Your task is to fill in a program that sums up some elements in a 3×3 matrix, $\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$. An input file specifies the elements to include. In this task and only in this task, you may assume that the input file is correct.

- **Input format:** Four ASCII-encoded digits:
  - i_0, the starting row index.
  - j_0, the starting column index.
  - h, the height.
  - w, the width.

  These numbers satisfy that $0 \leq i_0 < i_0 + h \leq 3$ and that $0 \leq j_0 < j_0 + w \leq 3$. Row and column indices start from 0.
- **Output format:** A byte that contains the sum of the elements between rows $i_0 \ldots (i_0 + h - 1)$ and columns $j_0 \ldots (j_0 + w - 1)$.

Figure 1 presents sample input and output. The program sums up all elements in the last two columns of the matrix: $1 + 2 + 4 + 5 + 7 + 8 = 27$. The `print` statement displays 27 as "\27".

```
0132
```

(a) Sample input

```
\27
```

(b) Sample output

Figure 1: Sample input and output

Finished reading at _____ : _____ : _____

Please implement this program in file "prog1.stu", **save** the file, and run the program by entering "`run prog1.stu`" in the terminal. If there is no response, press Ctrl+C in the terminal and check if your program contains an infinite loop.

File "input1.txt" contains the sample input. Please test your program as you normally would.

Finished implementing at _____ : _____ : _____

After your implementation, please read the **sample solution** for this problem.

4.2 Converting ASCII Integers

Started reading at _____ : _____ : _____

Your task is to write a program that parses an integer preceding the first space character on each line of a file. Your program should be able to handle arbitrary inputs by skipping malformed lines.

- **Input format:** Each line starts with an ASCII-encoded, variable-length integer followed by a space character.
- **Output format:** Each output byte has the value of the leading integer from each input line.

Figure 2 presents sample input and output. Table 1 explains the sample output.

```
65␣abc
66abc
67
68␣
```

(a) Sample input

```
AD
```

(b) Sample output

Figure 2: Sample input and output

Table 1: Explanation for sample output

Input line	Output byte	Explanation
1	65	"`print(65);`" displays `'A'`
2	(skipped)	unexpected character `'a'`
3	(skipped)	the trailing space character is missing
4	68	"`print(68);`" displays `'D'`

Finished reading at _____ : _____ : _____

Please implement this program in file "prog2.stu", **save** the file, and run the program by entering "`run prog2.stu`" in the terminal. If there is no response, press Ctrl+C in the terminal and check if your program contains an infinite loop.

File "input2.txt" contains the sample input. Please test your program as you normally would.

Finished implementing at _____ : _____ : _____

After your implementation, please read the **sample solution** for this problem.

4.3 Image Thumbnail Generator

Started reading at _____ : _____ : _____

Your task is to write a program that generates thumbnails for multiple bitmap images in a file. Your program should be able to handle arbitrary inputs by skipping malformed images.

Image thumbnails are small images that capture the overall shape of large images. For example, a thumbnail for Figure 3 is shown in Figure 4.

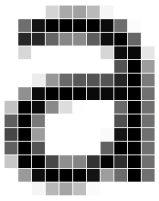


Figure 3: Magnified original image

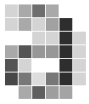


Figure 4: Magnified thumbnail

You are going to use the following algorithm that averages neighboring pixel values. Given a scaling factor s, the thumbnail for an image of height h and width w has height $\lfloor h/s \rfloor$ and width $\lfloor w/s \rfloor$. The value of the pixel in row i and column j of the thumbnail is the floor average of the values of all pixels in the s^2 square area between rows $i \cdot s \ldots (i \cdot s + s - 1)$ and columns $j \cdot s \ldots (j \cdot s + s - 1)$ of the original image.

- **Input format:** Each line describes an original image. Each line contains the following contents separated by a single space character:
  - An image name, which is a string of 1–10 characters long.
  - An ASCII-encoded variable-length integer, s, that represents the scaling factor.
  - An ASCII-encoded variable-length integer, h, that represents the original image height.
  - An ASCII-encoded variable-length integer, w, that represents the original image width.
  - $h \cdot w$ consecutive digits that each represents a pixel value. The pixels are ordered as follows: inside each row, pixels are ordered from left to right; the rows are each grouped together and ordered from top to bottom. Each pixel has one of ten possible values: '0'...'9'.
- **Output format:** The output contains lines that each describes a thumbnail. Each line contains the following contents separated by a single space character:
  - The image name.
  - $\lfloor h/s \rfloor \cdot \lfloor w/s \rfloor$ consecutive digits that each represents a pixel value.

Figure 5 presents sample input and output. Table 2 explains the sample output.

```
Img1␣2␣2␣2␣1234
Img2␣2␣4␣4␣1234567890123456
Img3␣2␣1␣2␣12
Img4␣3␣3␣4␣123456789012
```

(a) Sample input

```
Img1␣2
Img2␣3543
Img3␣
Img4␣3
```

(b) Sample output

Figure 5: Sample input and output

Table 2: Explanation for sample output

Name	Original	Thumbnail	Explanation
Img1	`1 2` `3 4`	`2`	With $s = 2$, the thumbnail for Img1 has height $\lfloor 2/2 \rfloor = 1$ and width $\lfloor 2/2 \rfloor = 1$. The single pixel has the value of $\lfloor (1+2+3+4)/4 \rfloor = \lfloor 10/4 \rfloor = 2$.

Img2	`1 2 3 4`
	`5 6 7 8`
	`9 0 1 2`
	`3 4 5 6`
		`3 5`
		`4 3`
			With $s = 2$, the thumbnail for Img2 has height $\lfloor 4/2 \rfloor = 2$ and width $\lfloor 4/2 \rfloor = 2$. The top-left pixel in the thumbnail has the value of $\lfloor (1+2+5+6)/4 \rfloor = 3$. The top-right pixel in the thumbnail has the value of $\lfloor (3+4+7+8)/4 \rfloor = 5$. The bottom-left pixel in the thumbnail has the value of $\lfloor (9+0+3+4)/4 \rfloor = 4$. The bottom-right pixel in the thumbnail has the value of $\lfloor (1+2+5+6)/4 \rfloor = 3$.

Img3	`1 2`	(empty)	With $s = 2$, the thumbnail for Img3 has height $\lfloor 1/2 \rfloor = 0$. This height indicates that the thumbnail image is empty.

Img4	`1 2 3 4`
	`5 6 7 8`
	`9 0 1 2`
		`3`	With $s = 3$, the thumbnail for Img4 has height $\lfloor 3/3 \rfloor = 1$ and width $\lfloor 4/3 \rfloor = 1$. The single pixel has the value of $\lfloor (1+2+3+5+6+7+9+0+1)/9 \rfloor = \lfloor 34/9 \rfloor = 3$.

Finished reading at _____ : _____ : _____

Please implement this program in file "prog3.stu", **save** the file, and run the program by entering "`run prog3.stu`" in the terminal. If there is no response, press Ctrl+C in the terminal and check if your program contains an infinite loop.

File "input3.txt" contains the sample input. Please test your program as you normally would.

Finished implementing at _____ : _____ : _____

5 Survey

5.1 Summing Matrix Elements

Mental Demand. How mentally demanding was task 1?
Please rate from 0 (very low) to 10 (very high).

Temporal Demand. How hurried or rushed was the pace of task 1?
Please rate from 0 (very low) to 10 (very high).

Performance. How successful were you in accomplishing task 1?
Please rate from 0 (failure) to 10 (perfect).

Frustration. How insecure, discouraged, irritated, stressed, and annoyed were you?
Please rate from 0 (very low) to 10 (very high).

How many times did you cheat? Where did you cheat?
Don't worry, there is no punishment for cheating.

Any other comments or clarification?

5.2 Converting ASCII Integers

Did the `inspect` construct help you with task 2?
Please circle: very helpful somewhat helpful neutral somewhat useless very useless

Mental Demand. How mentally demanding was task 2?
Please rate from 0 (very low) to 10 (very high).

Temporal Demand. How hurried or rushed was the pace of task 2?
Please rate from 0 (very low) to 10 (very high).

Performance. How successful were you in accomplishing task 2?
Please rate from 0 (failure) to 10 (perfect).

Frustration. How insecure, discouraged, irritated, stressed, and annoyed were you?
Please rate from 0 (very low) to 10 (very high).

How many times did you cheat? Where did you cheat?
Don't worry, there is no punishment for cheating.

Any other comments or clarification?

5.3 Image Thumbnail Generator

Did the `inspect` construct help you with task 3?
Please circle: very helpful somewhat helpful neutral somewhat useless very useless

Mental Demand. How mentally demanding was task 3?
Please rate from 0 (very low) to 10 (very high).

Temporal Demand. How hurried or rushed was the pace of task 3?
Please rate from 0 (very low) to 10 (very high).

Performance. How successful were you in accomplishing task 3?
Please rate from 0 (failure) to 10 (perfect).

Frustration. How insecure, discouraged, irritated, stressed, and annoyed were you?
Please rate from 0 (very low) to 10 (very high).

Any other comments or clarification?

6 Submission

1. Stop the screen recorder by clicking the blue monitor icon on the left and then clicking "Stop" in the window named "vokoscreen 1.9.0".
2. Shut down the virtual machine by clicking the gadget icon on the top-right corner, the "Shut Down..." option at the bottom of the drop-down menu, and the "Shut Down" icon on the right. DO NOT shut down from the VMware menu.
3. Send your updated file "LanguageFeaturesVM.vmwarevm" back to the experimenter.
4. Turn in this document.

B.3 Inspect Group Tutorials, Part I—The `inspect` Construct

The `inspect` Construct

An `inspect` loop is an iterator that automatically avoids errors. The syntax is as follows.

```
inspect (input_file, delimiter) {
  // Execute these statements on each input unit that allows successful execution.
  // These statements may contain function calls.
}
```

To illustrate the usage, we first present a small example. Say that we would like to collect the leading letters from all lines in a document. We would like to process the lines that start with alphabetic letters and to ignore the lines that start with other characters. Figure 1 presents sample input and output. Figure 2 compares two solutions, where Figure 2a uses traditional constructs and Figure 2b uses the `inspect` construct.

The program in Figure 2b works as follows. It first associates `f` to an input file named "data". The file consists of smaller input units, specifically, lines. The program starts execution from the `main` function, where an `inspect` loop iterates over the lines in file `f` until reaching the end of the file. For each line, the `inspect` loop automatically determines whether or not to execute the loop body based on whether or not the execution would succeed. Specifically,

- If a line starts with an alphabetic letter, the `inspect` loop executes the body which prints this leading letter.
- If a line does not start with an alphabetic letter, the `inspect` loop does not execute the body, so that there is no assertion failure.

In general, `inspect` loops iterate over delimited input units, skipping the units that would trigger errors. Consequently, each input unit is either successfully processed or completely ignored. The semantics use the following two criteria.

- **Delimiters:** A specified delimiter decomposes the input into smaller input units. An input unit is the contents between two delimiters. Each time the `inspect` body starts, a fresh input unit is available to be read.
- **Errors:** The `inspect` loop automatically skips the input units that would trigger errors if processed. Errors include assertion violations, arithmetic errors, array-access errors, file-access errors, attempts to read beyond input units, and resource exhaustion.

For completeness, please refer to other language constructs in a separate manual.

```
Hello,
world
!
```

(a) Sample input

```
Hw
```

(b) Sample output

Figure 1: Sample input and output

```
f = open("data");
main {
  while (!end(f)) {
    x = read(f);
    if ((x>='a' && x<='z') || (x>='A' && x<='Z')) {
      print(x);
    }
    while (!end(f) && x!='\n') {
      x = read(f);
    }
  }
  return 0;
}
```

(a) Traditional solution

```
f = open("data");
main {
  inspect (f, '\n') {
    x = read(f);
    assert((x>='a' && x<='z') || (x>='A' && x<='Z'));
    print(x);
  }
  return 0;
}
```

(b) `inspect` solution

Figure 2: Solutions

B.4 Inspect Group Tutorials, Part II—Other Language Constructs

Language Manual

This manual presents a subset of a programming language that contains simplified C constructs. For quick references, Tables 1, 2, and 3 list the sections that explain basic syntax, data operations, and control structures, respectively.

1

Table 1: Basic syntax

Keyword	Description	Section
	formatting	1.1
	comments	1.2
	variables	1.3
`print`	output an integer value	1.4
`assert`	assert an expression to be “true” (nonzero)	1.5

Table 2: Data operations

Keyword	Description	Section
	integer expressions	2.1
	array accesses	2.2
`malloc`	allocate memory for an array	2.2
`free`	deallocate memory from an array	2.2
`open`	open an input file	2.3
`read`	read a byte from an input file	2.3
`end`	test the end of the current input unit or input file	2.3
`seek`	update the input offset	2.3
`pos`	return current input offset	2.3

Table 3: Control structures

Keyword	Description	Section
`if`/`if-else`	conditional statements	3.1
`while`	loops	3.2
`break`	exit the current loop	3.2
`continue`	skip the current loop iteration	3.2
`main`	define the main function where a program starts	3.3
`func`	define a user function	3.4
`return`	return a value for a function	3.4

1 Basic syntax

1.1 Formatting

- **Statements:** Each statement terminates with a semicolon "`;`".
- **Spacing:** Code indentation does not matter. Arbitrary white spaces are allowed between different components.

1.2 Comments

A comment starts with "`//`" and contains all text up to the end of the line.

1.3 Variables

- **Naming:** Variable names must start with a letter and may contain the following characters: lowercase letters "a"... "z", uppercase letters "A"... "Z", digits "1"... "9", and underscore "_".
- **Assignment:** Figure 1 presents the syntax for assignment statements.
- **Types:** Each variable may belong to one of three types: integer, array, or file. The type of each variable remains unchanged during its lifetime.
- **Definition:** To define a new variable, assign it a value of the desired type.
- **Scoping:** Integer and array variables are local and must be defined inside functions. File variables are global and must be defined before all functions.

Example: Figure 2 presents an example program that defines variables. The program defines a file variable `f` for an input file named "data", an integer variable `i` of value `7`, and an array `a` consisting of 12 integers.

1.4 System output

Figure 3 presents the syntax for `print` statements. A `print` statement takes the lowest byte (8 bits) of the argument, converts the byte to display form, and prints it on the screen. Table 4 describes the display form.

1.5 Assertions

Figure 4 presents the syntax for `assert` statements. An `assert` statement triggers an error when the specified condition is "false" (evaluates to zero).

```
x = some_expression;
// x becomes the result of some_expression.
```

Figure 1: Syntax for variable assignments

```
f = open("data");
main {
  i = 7;
  a = malloc(12);
  // Do something here.
  return 0;
}
```

Figure 2: Example variable assignments

```
print(some_expression);
// Output the lowest byte in the result of some_expression.
```

Figure 3: Syntax for system output

```
assert(condition);
// If condition is false, trigger an error.
// If condition is true, this is a no-op.
```

Figure 4: Syntax for assertions

Table 4: System output display form

Byte value	Display form
9	horizontal tab (`'\t'`)
10	new line (`'\n'`)
32	space (`' '`)
33 . . . 126	the ASCII character for the value
others	a backslash "\" followed by the value

2 Data operations

2.1 Integers

- **Representation:** All integers are 32-bit two's-complement integers.
- **Boolean conversion:** When using integers in logical expressions, value `0` is "false" and non-zero values are "true". When using logical expressions in arithmetic operations, value "false" is `0` and value "true" is `1`.
- **Operators:** Tables 5, 6, and 7 describe arithmetic operators, logical operators, and brackets, respectively. Table 8 lists the precedence and associativity of these operators in descending precedence. There are no self-modifying operators such as "`++`" and "`+=`" in C.

2.2 Arrays

- **Allocation and deallocation:** Figure 5 presents the syntax for array allocation and deallocation. To allocate an array, the `malloc` statement requests space of a given size from the system memory, initializes all elements to `0`, and assigns the space to an array variable. To deallocate an existing array, the `free` statement returns its space to the system for future allocation and makes the array variable invalid.
- **Reading and writing:** To read or write an element in an array, use a pair of brackets "`[]`" to surround the index of the element. The indices range from `0` to `(n-1)` for arrays with `n` elements. There are no multi-dimensional arrays as in C.

Example: Figure 6 presents example code for array operations. The code allocates an array `a` with two integers, updates the elements to `1 0`, updates variable `x` to 0, and deallocates array `a`.

2.3 Files

- **Opening:** Figure 7a presents the syntax for opening files. An `open` statement associates an input file to a file variable and initializes the input offset to the start of the file. The `open` statement must appear outside any functions.
- **Reading:** Figure 7b presents the syntax for reading files. A `read` statement returns a byte from an input file and advances the input offset by one. The receiving variable to the left is mandatory.
- **Testing the end:** The `end` predicate tests whether all bytes in the current input unit or input file has been read.
- **Seeking:** Figure 7c presents the syntax for seeking input files to specific offsets.
- **Current offset:** The `pos` expression returns the current offset of a file.

Examples: Figure 8a presents an example program that outputs the contents in file "data". Figure 8b presents example code that seeks to the previous byte in a file.

Table 5: Arithmetic operators

Operator	Operation	Example expression	Example result
`+`	add	`9 + 4`	`13`
`-` (binary)	subtract	`9 - 4`	`5`
`-` (unary)	negation	`-4`	`-4` (0xfffffffc)
`*`	multiply	`9 * 4`	`36`
`/`	divide	`9 / 4`	`2`
`%`	modulo (remainder)	`9 % 4`	`1`
`&`	bitwise AND	`9 & 5`	`1` (0x00000001)
`|`	bitwise OR	`9 | 5`	`13` (0x0000000d)
`~`	bitwise NOT	`~9`	`-10` (0xfffffff6)
`>>`	shift right arithmetic	`15 >> 2`	`3` (0x00000003)
`<<`	shift left	`15 << 2`	`60` (0x0000003c)

Table 6: Logical operators

Operator	Operation	Example expression	Example result
`==`	equal to	`9 == 4`	`0` (“false”)
`!=`	unequal to	`9 != 4`	`1` (“true”)
`<`	less than	`9 < 4`	`0` (“false”)
`<=`	less than or equal to	`9 <= 4`	`0` (“false”)
`>`	greater than	`9 > 4`	`1` (“true”)
`>=`	greater than or equal to	`9 >= 4`	`1` (“true”)
`&&`	logical AND	`(1 < 2) && (3 < 2)`	`0` (“false”)
`||`	logical OR	`(1 < 2) || (3 < 2)`	`1` (“true”)
`!`	logical NOT	`!(1 < 2)`	`0` (“false”)

Table 7: Brackets

Operator	Operation
`()`	force precedence
`[]`	access an array

Table 8: Operator precedence and associativity

Precedence	Operator	Associativity
1	`()` `[]`	left to right
2	`-` (unary) `~`	N/A
3	`*` `/` `%`	left to right
4	`&` `|` `>>` `<<`	left to right
5	`+` `-` (binary)	left to right
6	`==` `!=` `<` `<=` `>` `>=`	left to right
7	`!`	N/A
8	`&&` `||`	left to right

```
arr = malloc(n);
// If the allocation succeeds, arr becomes a valid array of n elements initialized to 0.
```

(a) Allocation

```
// arr was a valid array.
free(arr);
// arr becomes invalid.
```

(b) Deallocation

Figure 5: Syntax for arrays

```
a = malloc(2); // a initializes to 0 0.
a[0] = 1; // a becomes 1 0.
x = a[1]; // x becomes 0.
free(a); // a becomes invalid.
```

Figure 6: Example array operations

```
f = open("data");
// If opening "data" succeeds, f becomes valid and is ready to read its 0th byte.
// Define functions here.
```

(a) Opening

```
// f was ready to read the i-th byte.
x = read(f);
// If reading succeeds, x becomes the value of the i-th byte, and f is ready to read
// the (i+1)-th byte.
```

(b) Reading

```
seek(f, x);
// f is ready to read the x-th byte.
```

(c) Seeking

Figure 7: Syntax for files

```
f = open("data");
main {
  while (!end(f)) {
    x = read(f);
    print(x);
  }
  return 0;
}
```

(a) Sequential accessing

```
seek(f, pos(f)-1);
```

(b) Random accessing

Figure 8: Example file operations

3 Control structures

3.1 Conditional statements

Figure 9 presents the syntax for two forms of conditional statements: `if` and `if-else`. The curly braces "`{}`" are mandatory. There are no shorthanded "`else if`" structures as in C.

Example: Figure 10 presents example code that uses an `if` statement to set variable `i` to `5`.

3.2 Loops

Figure 11 presents the syntax for `while` loops. A `while` loop repeats executing a block of code as long as a given condition holds. The curly braces "`{}`" are mandatory. There are no "`for`" loops as in C.

To manipulate the execution of `while` loops, `break` and `continue` statements exit the innermost surrounding loop and stop the current iteration of the innermost surrounding loop, respectively.

Examples: Figure 12 presents two pieces of example code that use `while` loops. Figure 12a is a no-op. Figure 12b contains an infinite loop.

3.3 The `main` function

Every program begins execution inside a special `main` function. Figure 13 presents the syntax for the `main` function.

3.4 User functions

- **Definition:** Figure 14a presents the syntax for the `func` keyword which defines user functions. Each user function may take an integer argument that is passed by value or may take no argument. Global files are accessible inside functions. Each function returns an integer value.
- **Naming:** Function names have the same rule as variable names discussed in Section 1.3.
- **Scoping:** All functions are global and must be defined in the top level of the program, parallel to the `main` function.
- **Invocation:** Figure 14b presents the syntax for calling user functions. There must be a variable that receives the return value.
- **Recursion:** User functions may be recursive, expressing operations by calling themselves.

Example: Figure 15 presents an example function, `inc`, which calculates `x` plus one.

```
if (condition) {
  // Execute if condition is true.
}
```

(a) `if`

```
if (condition) {
  // Execute if condition is true.
} else {
  // Execute if condition is false.
}
```

(b) `if-else`

Figure 9: Syntax for conditional statements

```
i = 2 - 7; // i becomes -5.
if (i < 0) { // Condition -5 < 0 is true.
  i = -i; // i becomes 5.
}
```

Figure 10: Example conditional statement

```
while (condition) {
  // Repeat executing while condition is true.
}
```

Figure 11: Syntax for `while` loops

```
while (i < 5) {
  break;
  i = i + 1;
}
```

(a) No-op

```
i = 0;
while (i < 5) {
  continue;
  i = i + 1;
}
```

(b) Infinite loop

Figure 12: Example `while` loops

```
// Open input files here.
// Define other functions here.
main {
  // Do something here.
  return some_expression;
}
```

Figure 13: Syntax for the `main` function

```
// Open input files here.
func foo (x) {
  // The argument is passed by value.
  // Do something here.
  return some_expression;
}
func bar () {
  // Do something here.
  return some_expression;
}
// Define other functions here.
```

(a) Definition

```
y = foo(some_expression);
z = bar();
```

(b) Invocation

Figure 14: Syntax for user functions

```
func inc (x) {
  return x + 1;
}
```

Figure 15: Example function definition

B.5 Inspect Group Tutorials, Part III—Sample Solutions

Sample Solutions

1 Summing Matrix Elements

```
// Problem 1:  Summing matrix elements

f = open("input1.txt");
main {
  a = malloc(9);
  a[0] = 0; a[1] = 1; a[2] = 2;
  a[3] = 3; a[4] = 4; a[5] = 5;
  a[6] = 6; a[7] = 7; a[8] = 8;
  sum = 0;
  i0 = read(f); i0 = i0 - '0';
  //================================
  //======== YOUR CODE HERE ========
  // Hint:
  // * Read the code above.
  // * To extract the value of an ASCII digit, subtract '0'.
  // * To access row i column j of the matrix, use index i*3+j.
  // * To increment integer variable x by one, use x=x+1.
  //================================
  j0 = read(f); j0 = j0 - '0';
  r = read(f); r = r - '0';
  c = read(f); c = c - '0';
  i = i0;
  while (i < i0 + r) {
    j = j0;
    while (j < j0 + c) {
      sum = sum + a[i*3+j];
      j = j + 1;
    }
    i = i + 1;
  }
  //================================
  print(sum);
  free(a);
  return 0;
}
```

2 Converting ASCII Integers

```
// Problem 2:  Converting ASCII integers

f = open("input2.txt");

//================================
//======== YOUR CODE HERE ========
// Hint:
// * Note the mandatory curly braces for conditional statements and loops.
// * A possible algorithm to parse an ASCII number is as follows:
//       ...
//       num = 0;
//       while (...)  {
//         ...
//         num = num * 10 + c - '0';
//         ...
//       }
//       ...
//================================
main {
  inspect (f, '\n') {
    c = read(f);
    num = 0;
    while (c != ' ') {
      assert(c >= '0' && c <= '9');
      num = num * 10 + c - '0';
      c = read(f);
    }
    print(num);
  }
  return 0;
}
//================================
```

B.6 Control Group Instructions

Instructions

You are going write three programs using a new programming language. Please follow the instructions below to confirm your eligibility, setup the virtual machine, learn the language, complete the writing tasks, complete surveys regarding your experience, and submit your data.

1 Eligibility

In the **past five years**, you have _____ years' experience using imperative languages _____ (such as C/C++, Java, or Python).

2 Environmental setup

1. Launch the virtual machine by opening the file "LanguageFeaturesVM.vmwarevm" with VMware Fusion.
2. Adjust the VMware window to your screen, so that you can see both the lower-left corner with a trash-can icon and the upper-right corner with a gadget icon.
3. Launch the screen recorder by clicking the blue monitor icon on the left. A window titled "vokoscreen 1.9.0" should appear. [1]
4. Click "Start" in the vokoscreen window. The vokoscreen window should minimize automatically, and there should be a small white triangle to the left of the blue monitor icon that you have previously clicked.
5. Launch a terminal by clicking the black rectangle icon on the left. A terminal window titled "default@ubuntu:~\Desktop" should appear.

[1] For analyzing purposes, we record the screen in the virtual machine as you perform tasks in this experiment. We have configured vokoscreen to record the full screen, without audio, capturing 25 frames per second, encoding in the x264 format, storing into an avi file, and including mouse cursor movements. Feel free to double check these configurations and ask the experimenter if you have any questions.

3 Language tutorial

The language used in this experiment is similar to many other imperative languages such as C.

1. Learn about the language by reading the *Language Manual*.
2. **Double-click** the file "prog0.stu" on the desktop. An editor window titled "prog0.stu (~\Desktop) – gedit" should appear. This file contains an example program that counts the number of characters on each line of the file "input0.txt".
3. Run this example program by entering

   ```
   run prog0.stu
   ```

   in the terminal. You should see

   ```
   [Program return value]: 0.
   \5
   \3\0
   ```

   in the same terminal window. The first line presents the value returned from the program's `main` function. Then follows the outputs from `print` statements. The strings "\5", "\3", and "\0" indicate that there are 5, 3, and 0 characters on the first, the third, and the fourth lines, respectively. The newline character between "\5" and "\3" indicates that there are 10 characters on the second line, since the number 10 corresponds to the newline character `'\n'` in ASCII encoding.
4. Feel free to edit files "prog0.stu" and "input0.txt" and run the program again.

4 Writing tasks

There are three writing tasks. The first two tasks should help you complete the third task. Please complete them in order, and fill in the time that you start and finish each task.

4.1 Summing Matrix Elements

Started reading at _____ : _____ : _____

Your task is to fill in a program that sums up some elements in a 3×3 matrix, $\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{pmatrix}$. An input file specifies the elements to include. In this task and only in this task, you may assume that the input file is correct.

- **Input format:** Four ASCII-encoded digits:
  - i_0, the starting row index.
  - j_0, the starting column index.
  - h, the height.
  - w, the width.

  These numbers satisfy that $0 \leq i_0 < i_0 + h \leq 3$ and that $0 \leq j_0 < j_0 + w \leq 3$. Row and column indices start from 0.
- **Output format:** A byte that contains the sum of the elements between rows $i_0 \ldots (i_0 + h - 1)$ and columns $j_0 \ldots (j_0 + w - 1)$.

Figure 1 presents sample input and output. The program sums up all elements in the last two columns of the matrix: $1 + 2 + 4 + 5 + 7 + 8 = 27$. The `print` statement displays 27 as "\27".

```
0132
```

(a) Sample input

```
\27
```

(b) Sample output

Figure 1: Sample input and output

Finished reading at _____ : _____ : _____

Please implement this program in file "prog1.stu", **save** the file, and run the program by entering "`run prog1.stu`" in the terminal. If there is no response, press Ctrl+C in the terminal and check if your program contains an infinite loop.

File "input1.txt" contains the sample input. Please test your program as you normally would.

Finished implementing at _____ : _____ : _____

After your implementation, please read the **sample solution** for this problem.

4.2 Converting ASCII Integers

Started reading at _____ : _____ : _____

Your task is to write a program that parses an integer preceding the first space character on each line of a file. Your program should be able to handle arbitrary inputs by skipping malformed lines.

- **Input format:** Each line starts with an ASCII-encoded, variable-length integer followed by a space character.
- **Output format:** Each output byte has the value of the leading integer from each input line.

Figure 2 presents sample input and output. Table 1 explains the sample output.

```
65␣abc
66abc
67
68␣
```

(a) Sample input

```
AD
```

(b) Sample output

Figure 2: Sample input and output

Table 1: Explanation for sample output

Input line	Output byte	Explanation
1	65	"`print(65);`" displays `'A'`
2	(skipped)	unexpected character `'a'`
3	(skipped)	the trailing space character is missing
4	68	"`print(68);`" displays `'D'`

Finished reading at _____ : _____ : _____

Please implement this program in file "prog2.stu", **save** the file, and run the program by entering "`run prog2.stu`" in the terminal. If there is no response, press Ctrl+C in the terminal and check if your program contains an infinite loop.

File "input2.txt" contains the sample input. Please test your program as you normally would.

Finished implementing at _____ : _____ : _____

After your implementation, please read the **sample solution** for this problem.

4.3 Image Thumbnail Generator

Started reading at _____ : _____ : _____

Your task is to write a program that generates thumbnails for multiple bitmap images in a file. Your program should be able to handle arbitrary inputs by skipping malformed images.

Image thumbnails are small images that capture the overall shape of large images. For example, a thumbnail for Figure 3 is shown in Figure 4.

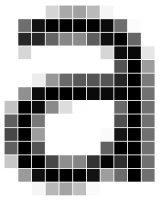


Figure 3: Magnified original image

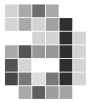


Figure 4: Magnified thumbnail

You are going to use the following algorithm that averages neighboring pixel values. Given a scaling factor s, the thumbnail for an image of height h and width w has height $\lfloor h/s \rfloor$ and width $\lfloor w/s \rfloor$. The value of the pixel in row i and column j of the thumbnail is the floor average of the values of all pixels in the s^2 square area between rows $i \cdot s \ldots (i \cdot s + s - 1)$ and columns $j \cdot s \ldots (j \cdot s + s - 1)$ of the original image.

- **Input format:** Each line describes an original image. Each line contains the following contents separated by a single space character:
  - An image name, which is a string of 1–10 characters long.
  - An ASCII-encoded variable-length integer, s, that represents the scaling factor.
  - An ASCII-encoded variable-length integer, h, that represents the original image height.
  - An ASCII-encoded variable-length integer, w, that represents the original image width.
  - $h \cdot w$ consecutive digits that each represents a pixel value. The pixels are ordered as follows: inside each row, pixels are ordered from left to right; the rows are each grouped together and ordered from top to bottom. Each pixel has one of ten possible values: '0'...'9'.
- **Output format:** The output contains lines that each describes a thumbnail. Each line contains the following contents separated by a single space character:
  - The image name.
  - $\lfloor h/s \rfloor \cdot \lfloor w/s \rfloor$ consecutive digits that each represents a pixel value.

Figure 5 presents sample input and output. Table 2 explains the sample output.

```
Img1␣2␣2␣2␣1234
Img2␣2␣4␣4␣1234567890123456
Img3␣2␣1␣2␣12
Img4␣3␣3␣4␣123456789012
```

(a) Sample input

```
Img1␣2
Img2␣3543
Img3␣
Img4␣3
```

(b) Sample output

Figure 5: Sample input and output

Table 2: Explanation for sample output

Name	Original	Thumbnail	Explanation
Img1	`1 2` `3 4`	`2`	With $s = 2$, the thumbnail for Img1 has height $\lfloor 2/2 \rfloor = 1$ and width $\lfloor 2/2 \rfloor = 1$. The single pixel has the value of $\lfloor (1+2+3+4)/4 \rfloor = \lfloor 10/4 \rfloor = 2$.

Img2	`1 2 3 4`
	`5 6 7 8`
	`9 0 1 2`
	`3 4 5 6`
		`3 5`
		`4 3`
			With $s = 2$, the thumbnail for Img2 has height $\lfloor 4/2 \rfloor = 2$ and width $\lfloor 4/2 \rfloor = 2$. The top-left pixel in the thumbnail has the value of $\lfloor (1+2+5+6)/4 \rfloor = 3$. The top-right pixel in the thumbnail has the value of $\lfloor (3+4+7+8)/4 \rfloor = 5$. The bottom-left pixel in the thumbnail has the value of $\lfloor (9+0+3+4)/4 \rfloor = 4$. The bottom-right pixel in the thumbnail has the value of $\lfloor (1+2+5+6)/4 \rfloor = 3$.

Img3	`1 2`	(empty)	With $s = 2$, the thumbnail for Img3 has height $\lfloor 1/2 \rfloor = 0$. This height indicates that the thumbnail image is empty.

Img4	`1 2 3 4`
	`5 6 7 8`
	`9 0 1 2`
		`3`	With $s = 3$, the thumbnail for Img4 has height $\lfloor 3/3 \rfloor = 1$ and width $\lfloor 4/3 \rfloor = 1$. The single pixel has the value of $\lfloor (1+2+3+5+6+7+9+0+1)/9 \rfloor = \lfloor 34/9 \rfloor = 3$.

Finished reading at _____ : _____ : _____

Please implement this program in file "prog3.stu", **save** the file, and run the program by entering "`run prog3.stu`" in the terminal. If there is no response, press Ctrl+C in the terminal and check if your program contains an infinite loop.

File "input3.txt" contains the sample input. Please test your program as you normally would.

Finished implementing at _____ : _____ : _____

5 Survey

5.1 Summing Matrix Elements

Mental Demand. How mentally demanding was task 1?
Please rate from 0 (very low) to 10 (very high).

Temporal Demand. How hurried or rushed was the pace of task 1?
Please rate from 0 (very low) to 10 (very high).

Performance. How successful were you in accomplishing task 1?
Please rate from 0 (failure) to 10 (perfect).

Frustration. How insecure, discouraged, irritated, stressed, and annoyed were you?
Please rate from 0 (very low) to 10 (very high).

How many times did you cheat? Where did you cheat?
Don't worry, there is no punishment for cheating.

Any other comments or clarification?

5.2 Converting ASCII Integers

Mental Demand. How mentally demanding was task 2?
Please rate from 0 (very low) to 10 (very high).

Temporal Demand. How hurried or rushed was the pace of task 2?
Please rate from 0 (very low) to 10 (very high).

Performance. How successful were you in accomplishing task 2?
Please rate from 0 (failure) to 10 (perfect).

Frustration. How insecure, discouraged, irritated, stressed, and annoyed were you?
Please rate from 0 (very low) to 10 (very high).

How many times did you cheat? Where did you cheat?
Don't worry, there is no punishment for cheating.

Any other comments or clarification?

5.3 Image Thumbnail Generator

Mental Demand. How mentally demanding was task 3?
Please rate from 0 (very low) to 10 (very high).

Temporal Demand. How hurried or rushed was the pace of task 3?
Please rate from 0 (very low) to 10 (very high).

Performance. How successful were you in accomplishing task 3?
Please rate from 0 (failure) to 10 (perfect).

Frustration. How insecure, discouraged, irritated, stressed, and annoyed were you?
Please rate from 0 (very low) to 10 (very high).

Any other comments or clarification?

6 Submission

1. Stop the screen recorder by clicking the blue monitor icon on the left and then clicking "Stop" in the window named "vokoscreen 1.9.0".
2. Shut down the virtual machine by clicking the gadget icon on the top-right corner, the "Shut Down..." option at the bottom of the drop-down menu, and the "Shut Down" icon on the right. DO NOT shut down from the VMware menu.
3. Send your updated file "LanguageFeaturesVM.vmwarevm" back to the experimenter.
4. Turn in this document.

B.7 Control Group Tutorials, Part I—Language

Language Manual

This manual presents a subset of a programming language that contains simplified C constructs. For quick references, Tables 1, 2, and 3 list the sections that explain basic syntax, data operations, and control structures, respectively.

1

Table 1: Basic syntax

Keyword	Description	Section
	formatting	1.1
	comments	1.2
	variables	1.3
`print`	output an integer value	1.4

Table 2: Data operations

Keyword	Description	Section
	integer expressions	2.1
	array accesses	2.2
`valid` (array)	test the validity of an array variable	2.2
`malloc`	allocate memory for an array	2.2
`free`	deallocate memory from an array	2.2
`valid` (file)	test the validity of a file variable	2.3
`open`	open an input file	2.3
`read`	read a byte from an input file	2.3
`end`	test the end of the input file	2.3
`seek`	update the input offset	2.3
`pos`	return current input offset	2.3

Table 3: Control structures

Keyword	Description	Section
`if`/`if-else`	conditional statements	3.1
`while`	loops	3.2
`break`	exit the current loop	3.2
`continue`	skip the current loop iteration	3.2
`main`	define the main function where a program starts	3.3
`func`	define a user function	3.4
`return`	return a value for a function	3.4

1 Basic syntax

1.1 Formatting

- **Statements:** Each statement terminates with a semicolon "`;`".
- **Spacing:** Code indentation does not matter. Arbitrary white spaces are allowed between different components.

1.2 Comments

A comment starts with "`//`" and contains all text up to the end of the line.

1.3 Variables

- **Naming:** Variable names must start with a letter and may contain the following characters: lowercase letters "a"... "z", uppercase letters "A"... "Z", digits "1"... "9", and underscore "_".
- **Assignment:** Figure 1 presents the syntax for assignment statements.
- **Types:** Each variable may belong to one of three types: integer, array, or file. The type of each variable remains unchanged during its lifetime.
- **Definition:** To define a new variable, assign it a value of the desired type.
- **Scoping:** Integer and array variables are local and must be defined inside functions. File variables are global and must be defined before all functions.

Example: Figure 2 presents an example program that defines variables. The program defines a file variable `f` for an input file named "data", an integer variable `i` of value `7`, and an array `a` consisting of 12 integers.

1.4 System output

Figure 3 presents the syntax for `print` statements. A `print` statement takes the lowest byte (8 bits) of the argument, converts the byte to display form, and prints it on the screen. Table 4 describes the display form.

```
x = some_expression;
// x becomes the result of some_expression.
```

Figure 1: Syntax for variable assignments

```
f = open("data");
main {
  i = 7;
  a = malloc(12);
  // Do something here.
  return 0;
}
```

Figure 2: Example variable assignments

```
print(some_expression);
// Output the lowest byte in the result of some_expression.
```

Figure 3: Syntax for system output

Table 4: System output display form

Byte value	Display form
9	horizontal tab (`'\t'`)
10	new line (`'\n'`)
32	space (`' '`)
33 . . . 126	the ASCII character for the value
others	a backslash "\" followed by the value

2 Data operations

2.1 Integers

- **Representation:** All integers are 32-bit two's-complement integers.
- **Boolean conversion:** When using integers in logical expressions, value `0` is "false" and non-zero values are "true". When using logical expressions in arithmetic operations, value "false" is `0` and value "true" is `1`.
- **Operators:** Tables 5, 6, and 7 describe arithmetic operators, logical operators, and brackets, respectively. Table 8 lists the precedence and associativity of these operators in descending precedence. There are no self-modifying operators such as "`++`" and "`+=`" in C.

2.2 Arrays

- **Validity:** The `valid` predicate tests whether an array variable is valid.
- **Allocation and deallocation:** Figure 4 presents the syntax for array allocation and deallocation. To allocate an array, the `malloc` statement requests space of a given size from the system memory, initializes all elements to `0`, and assigns the space to an array variable. To deallocate an existing array, the `free` statement returns its space to the system for future allocation and makes the array variable invalid.
- **Reading and writing:** To read or write an element in an array, use a pair of brackets "`[]`" to surround the index of the element. The indices range from `0` to `(n-1)` for arrays with `n` elements. There are no multi-dimensional arrays as in C.

Examples: Figure 5a presents example code for array accesses. The code allocates an array `a` with two integers, updates the elements to `1 0`, and updates variable `x` to 0. Figure 5b presents example code that safely deallocates an array that may be invalid.

2.3 Files

- **Validity:** The `valid` predicate tests whether a file variable is valid.
- **Opening:** Figure 6a presents the syntax for opening files. An `open` statement associates an input file to a file variable and initializes the input offset to the start of the file. The `open` statement must appear outside any functions.
- **Reading:** Figure 6b presents the syntax for reading files. A `read` statement returns a byte from an input file and advances the input offset by one. The receiving variable to the left is mandatory. The statement returns -1 if the file variable is invalid or its offset is invalid.
- **Testing the end:** The `end` predicate tests whether all bytes in the current input file has been read. The predicate returns -1 if the file variable is invalid.
- **Seeking:** Figure 6c presents the syntax for seeking input files to specific offsets. The statement returns -1 if the file variable is invalid or the offset is invalid.
- **Current offset:** The `pos` expression returns the current offset of a file. The expression returns -1 if the file variable is invalid.

Examples: Figure 7a presents an example program that outputs the contents in file "data". Figure 7b presents example code that seeks to the previous byte in a file.

Table 5: Arithmetic operators

Operator	Operation	Example expression	Example result
`+`	add	`9 + 4`	`13`
`-` (binary)	subtract	`9 - 4`	`5`
`-` (unary)	negation	`-4`	`-4` (0xfffffffc)
`*`	multiply	`9 * 4`	`36`
`/`	divide	`9 / 4`	`2`
`%`	modulo (remainder)	`9 % 4`	`1`
`&`	bitwise AND	`9 & 5`	`1` (0x00000001)
`|`	bitwise OR	`9 | 5`	`13` (0x0000000d)
`~`	bitwise NOT	`~9`	`-10` (0xfffffff6)
`>>`	shift right arithmetic	`15 >> 2`	`3` (0x00000003)
`<<`	shift left	`15 << 2`	`60` (0x0000003c)

Table 6: Logical operators

Operator	Operation	Example expression	Example result
`==`	equal to	`9 == 4`	`0` (“false”)
`!=`	unequal to	`9 != 4`	`1` (“true”)
`<`	less than	`9 < 4`	`0` (“false”)
`<=`	less than or equal to	`9 <= 4`	`0` (“false”)
`>`	greater than	`9 > 4`	`1` (“true”)
`>=`	greater than or equal to	`9 >= 4`	`1` (“true”)
`&&`	logical AND	`(1 < 2) && (3 < 2)`	`0` (“false”)
`||`	logical OR	`(1 < 2) || (3 < 2)`	`1` (“true”)
`!`	logical NOT	`!(1 < 2)`	`0` (“false”)

Table 7: Brackets

Operator	Operation
`()`	force precedence
`[]`	access an array

Table 8: Operator precedence and associativity

Precedence	Operator	Associativity
1	`()` `[]`	left to right
2	`-` (unary) `~`	N/A
3	`*` `/` `%`	left to right
4	`&` `|` `>>` `<<`	left to right
5	`+` `-` (binary)	left to right
6	`==` `!=` `<` `<=` `>` `>=`	left to right
7	`!`	N/A
8	`&&` `||`	left to right

```
arr = malloc(n);
// If the allocation succeeds, arr becomes a valid array of n elements initialized to 0.
```

(a) Allocation

```
// arr was a valid array.
free(arr);
// arr becomes invalid.
```

(b) Deallocation

Figure 4: Syntax for arrays

```
a = malloc(2); // a initializes to 0 0.
a[0] = 1; // a becomes 1 0.
x = a[1]; // x becomes 0.
```

(a) Accessing

```
// a may or may not be valid.
if (valid(a)) {
  free(a);
}
// a becomes invalid.
```

(b) Deallocating

Figure 5: Example array operations

```
f = open("data");
// If opening "data" succeeds, f becomes valid and is ready to read its 0th byte.
// Define functions here.
```

(a) Opening

```
// f was ready to read the i-th byte.
x = read(f);
// If reading succeeds, x becomes the value of the i-th byte, and f is ready to read
// the (i+1)-th byte.
// If reading fails, x becomes -1.
```

(b) Reading

```
y = seek(f, x);
// If seeking succeeds, y becomes x, and f is ready to read the x-th byte.
// If seeking fails, y becomes -1.
```

(c) Seeking

Figure 6: Syntax for files

```
f = open("data");
main {
  if (!valid(f)) {
    return -1;
  }
  while (!end(f)) {
    x = read(f);
    print(x);
  }
  return 0;
}
```

(a) Sequential accessing

```
y = seek(f, pos(f)-1);
```

(b) Random accessing

Figure 7: Example file operations

3 Control structures

3.1 Conditional statements

Figure 8 presents the syntax for two forms of conditional statements: `if` and `if-else`. The curly braces "`{}`" are mandatory. There are no shorthanded "`else if`" structures as in C.

Example: Figure 9 presents example code that uses an `if` statement to set variable `i` to `5`.

3.2 Loops

Figure 10 presents the syntax for `while` loops. A `while` loop repeats executing a block of code as long as a given condition holds. The curly braces "`{}`" are mandatory. There are no "`for`" loops as in C.

To manipulate the execution of `while` loops, `break` and `continue` statements exit the innermost surrounding loop and stop the current iteration of the innermost surrounding loop, respectively.

Examples: Figure 11 presents two pieces of example code that use `while` loops. Figure 11a is a no-op. Figure 11b contains an infinite loop.

3.3 The `main` function

Every program begins execution inside a special `main` function. Figure 12 presents the syntax for the `main` function.

3.4 User functions

- **Definition:** Figure 13a presents the syntax for the `func` keyword which defines user functions. Each user function may take an integer argument that is passed by value or may take no argument. Global files are accessible inside functions. Each function returns an integer value.
- **Naming:** Function names have the same rule as variable names discussed in Section 1.3.
- **Scoping:** All functions are global and must be defined in the top level of the program, parallel to the `main` function.
- **Invocation:** Figure 13b presents the syntax for calling user functions. There must be a variable that receives the return value.
- **Recursion:** User functions may be recursive, expressing operations by calling themselves.

Example: Figure 14 presents an example function, `inc`, which calculates `x` plus one.

```
if (condition) {
  // Execute if condition is true.
}
```

(a) `if`

```
if (condition) {
  // Execute if condition is true.
} else {
  // Execute if condition is false.
}
```

(b) `if-else`

Figure 8: Syntax for conditional statements

```
i = 2 - 7; // i becomes -5.
if (i < 0) { // Condition -5 < 0 is true.
  i = -i; // i becomes 5.
}
```

Figure 9: Example conditional statement

```
while (condition) {
  // Repeat executing while condition is true.
}
```

Figure 10: Syntax for `while` loops

```
while (i < 5) {
  break;
  i = i + 1;
}
```

(a) No-op

```
i = 0;
while (i < 5) {
  continue;
  i = i + 1;
}
```

(b) Infinite loop

Figure 11: Example `while` loops

```
// Open input files here.
// Define other functions here.
main {
  // Do something here.
  return some_expression;
}
```

Figure 12: Syntax for the `main` function

```
// Open input files here.
func foo (x) {
  // The argument is passed by value.
  // Do something here.
  return some_expression;
}
func bar () {
  // Do something here.
  return some_expression;
}
// Define other functions here.
```

(a) Definition

```
y = foo(some_expression);
z = bar();
```

(b) Invocation

Figure 13: Syntax for user functions

```
func inc (x) {
  return x + 1;
}
```

Figure 14: Example function definition

B.8 Control Group Tutorials, Part II—Sample Solutions

Sample Solutions

1 Summing Matrix Elements

```
// Problem 1:  Summing matrix elements

f = open("input1.txt");
main {
  a = malloc(9);
  a[0] = 0; a[1] = 1; a[2] = 2;
  a[3] = 3; a[4] = 4; a[5] = 5;
  a[6] = 6; a[7] = 7; a[8] = 8;
  sum = 0;
  i0 = read(f); i0 = i0 - '0';
  //================================
  //======== YOUR CODE HERE ========
  // Hint:
  // * Read the code above.
  // * To extract the value of an ASCII digit, subtract '0'.
  // * To access row i column j of the matrix, use index i*3+j.
  // * To increment integer variable x by one, use x=x+1.
  //================================
  j0 = read(f); j0 = j0 - '0';
  r = read(f); r = r - '0';
  c = read(f); c = c - '0';
  i = i0;
  while (i < i0 + r) {
    j = j0;
    while (j < j0 + c) {
      sum = sum + a[i*3+j];
      j = j + 1;
    }
    i = i + 1;
  }
  //================================
  print(sum);
  free(a);
  return 0;
}
```

2 Converting ASCII Integers

```
// Problem 2:  Converting ASCII integers

f = open("input2.txt");

//================================
//======== YOUR CODE HERE ========
// Hint:
// * Note the mandatory curly braces for conditional statements and loops.
// * A possible algorithm to parse an ASCII number is as follows:
//       ...
//       num = 0;
//       while (...)  {
//         ...
//         num = num * 10 + c - '0';
//         ...
//       }
//       ...
//================================
main {
  if (!valid(f)) {
    return -1;
  }
  while(!end(f)) {
    bad = 0;
    num = 0;
    c = read(f);
    while (c != ' ' && c != '\n') {
      if (c < '0' || c > '9') {
        bad = 1;
        break;
      }
      num = num * 10 + c - '0';
      c = read(f);
    }
    if (!bad && c == ' ') {
      print(num);
    }
    while (c != '\n' && !end(f)) {
      c = read(f);
    }
  }
  return 0;
}
//================================
```

C. Benchmark Programs for Other Input Formats

This appendix describes the benchmarks in detail and presents the source code for all the programs. Table 10 presents the language constructs available for the four versions of the benchmarks. More detail about these language constructs are presented in Appendix D.

C.1 CSV

CSV files store tables. A CSV file contains a title line followed by content lines. Each line contains fields that are separated by commas.

The benchmark program reads a CSV file, expecting that each field is at most 10 characters long, and that each content line has the same number of fields as the title line. The program outputs the fields and lines that do not trigger errors. In effect, the program discards fields that are longer than 10 characters, trailing fields that would otherwise make a line contain too many fields, and content lines that do not have enough fields.

C.1.1 Full RIFL Version

```
main {
  f = opent("../inputs/csv/newlines.csv");

  // titles
  columns = 0;
  inspectt (1, f, ',', '\n') {
    title = malloc(10);
    i = 0;
    while (!end(f)) {
      x = read(f);
      title[i] = x;
      i = i + 1;
    }
    if (columns > 0) {
      print(',');
    }
    print(title);
    free(title);
    columns = columns + 1;
  }
  print('\n');
  print('\n');

  // contents
  assert(columns > 0);
  inspectt (!end(f), f, '\n') {
    j = 0;
    inspectt (j < columns, f, ',') {
      field = malloc(10);
      i = 0;
      while (!end(f)) {
        x = read(f);
        field[i] = x;
        i = i + 1;
      }
      if (j > 0) {
        print(',');
      }
      print(field);
      free(field);
      j = j + 1;
    }
    print('\n');
    assert(j == columns);
  }
  return 0;
}
```

Table 10: Language restrictions for four versions

Versions	Loops	System calls
Full RIFL	inspectt, inspectb, lookatt, lookatb, while	assert, malloc, free, end, pos, opent, openb, seek, read
Explicit Check	inspectt, inspectb, lookatt, lookatb, while	assert, malloc_ec, free_ec, end_ec, pos_ec, opent_ec, openb_ec, seek_ec, read_ec
Explicit Recovery	lookatt, lookatb, while	malloc_ec, free_ec, end_ec, pos_ec, opent_ec, openb_ec, seek_ec, read_ec
Conventional	while	malloc_ec, free_ec, end_ec, pos_ec, opent_ec, openb_ec, seek_ec, read_ec

C.1.2 Explicit Check Version

```
main {
  f = opent_ec("../inputs/csv/newlines.csv");
  assert(valid(f));

  // titles
  columns = 0;
  inspectt (1, f, ',', '\n') {
    title = malloc_ec(10);
    assert(valid(title));
    i = 0;
    while (!end_ec(f)) {
      assert(i < 10);
      x = read_ec(f);
      assert(x >= 0);
      title[i] = x;
      i = i + 1;
    }
    if (columns > 0) {
      print(',');
    }
    print(title);
    x = free_ec(title);
    columns = columns + 1;
  }
  print('\n');
  print('\n');

  // contents
  assert(columns > 0);
  inspectt (!end_ec(f), f, '\n') {
    j = 0;
    inspectt (j < columns, f, ',') {
      field = malloc_ec(10);
      assert(valid(field));
      i = 0;
      while (!end_ec(f)) {
        assert(i < 10);
        x = read_ec(f);
        assert(x >= 0);
        field[i] = x;
        i = i + 1;
      }
      if (j > 0) {
        print(',');
      }
      print(field);
      x = free_ec(field);
      j = j + 1;
    }
    print('\n');
    assert(j == columns);
  }
```

```
  return 0;
}
```

C.1.3 Explicit Recovery Version

```
main {
  f = opent_ec("../inputs/csv/newlines.csv");
  if (!valid(f)) {
    exit(1);
  }

  // titles
  columns = 0;
  lookatt (1, f, ',', '\n') {
    title = malloc_ec(10);
    if (valid(title)) {
      bad = 0;
      i = 0;
      while (!end_ec(f) && i < 10) {
        x = read_ec(f);
        if (x < 0) {
          bad = 1;
        }
        title[i] = x;
        i = i + 1;
      }
      if (end_ec(f) && !bad) {
        if (columns > 0) {
          print(',');
        }
        print(title);
        columns = columns + 1;
      } // skip unit
      x = free_ec(title);
    } // skip unit
  }
  print('\n');
  print('\n');

  // contents
  if (columns <= 0) {
    exit(1);
  }
  buffer = malloc_ec(11 * columns);
  if (!valid(buffer)) {
    exit(1);
  }
  lookatt (!end_ec(f), f, '\n') {
    idx = 0;
    while (idx < 11 * columns) {
      buffer[idx] = 0;
      idx = idx + 1;
    }
    idx = 0;
    j = 0;
    lookatt (j < columns, f, ',') {
      start = idx;
      if (j > 0) {
        buffer[idx] = ',';
        idx = idx + 1;
      }
      i = 0;
      while (!end_ec(f) && x >= 0 && i < 10) {
        x = read_ec(f);
        buffer[idx] = x;
        idx = idx + 1;
        i = i + 1;
      }
      if (end_ec(f) && x >= 0) {
        j = j + 1;
      } else { // skip unit
        while (idx > start) {
          buffer[idx] = 0;
          idx = idx - 1;
        }
      }
    }
    if (j == columns) {
      print(buffer);
      print('\n');
    } // skip unit
  }
  x = free_ec(buffer);
  return 0;
}
```

C.1.4 Conventional Version

```
main {
  f = opent_ec("../inputs/csv/newlines.csv");
  if (!valid(f)) {
    exit(1);
  }

  // titles
  columns = 0;
  finish = 0;
  while (!finish) {
    title = malloc_ec(10);
    if (valid(title)) {
      bad = 0;
      i = 1;
      x = read_ec(f);
      while (!bad && x != ',' && x != '\n' && i <
          10) {
        if (x < 0) {
          bad = 1;
        }
        title[i] = x;
        i = i + 1;
        x = read_ec(f);
      }
      if (!bad && (x == ',' || x == '\n')) {
        if (columns > 0) {
          print(',');
        }
        print(title);
        columns = columns + 1;
      } // skip unit
      while (x >= 0 && x != ',' && x != '\n') {
        x = read_ec(f);
      }
      dummy = free_ec(title);
    } // skip unit
    if (x < 0 || x == '\n') {
      finish = 1;
    }
  }
  print('\n');
  print('\n');

  // contents
  if (columns <= 0) {
    exit(1);
  }
  buffer = malloc_ec(11 * columns);
  if (!valid(buffer)) {
    exit(1);
  }
  while (!end_ec(f)) {
    idx = 0;
    while (idx < 11 * columns) {
      buffer[idx] = 0;
      idx = idx + 1;
    }
    idx = 0;
    j = 0;
    x = 0;
    while (j < columns && x >= 0 && x != '\n') {
      start = idx;
      if (j > 0) {
        buffer[idx] = ',';
        idx = idx + 1;
      }
      i = 0;
      x = read_ec(f);
      while (x >= 0 && x != '\n' && x != ',' && i
          < 10) {
        buffer[idx] = x;
        idx = idx + 1;
        i = i + 1;
        x = read_ec(f);
```

```
        }
        if (x == '\n' || x == ',') {
          j = j + 1;
        } else { // skip unit
          while (idx > start) {
            buffer[idx] = 0;
            idx = idx - 1;
          }
        }
        while (x >= 0 && x != '\n' && x != ',') {
          x = read_ec(f);
        }
      }
      if (j == columns) {
        print(buffer);
        print('\n');
      } // skip unit
      while (x >= 0 && x != '\n') {
        x = read_ec(f);
      }
    }
    x = free_ec(buffer);
    return 0;
  }
```

C.2 OBJ

OBJ files express the shapes of objects. An OBJ file contains lines that each defines a vertex or a face. Each vertex definition contains the character `'v'` followed by three space-delimited decimal numbers that represent a three-dimensional position. Each face definition contains the character `'f'` followed by at least three space-delimited positive integers that each refer back to a vertex. Specifically, an integer i refers to the i-th vertex that the preceding inputs define.

The benchmark program reads an OBJ file, expecting that each decimal number is at most 10 characters long. The program also checks to ensure that each integer in a face definition is at most the number of existing vertices. The program outputs the definitions of vertices and faces that do not trigger errors. In effect, the program discards the malformed numbers, the vertices that do not contain exactly three dimensions, and the faces that do not have at least three valid vertex references.

C.2.1 Full RIFL Version

```
f = opent("../inputs/obj/icosahedron-2.obj");
num = malloc(10);
dim = 0;

func cleannum(dummy) {
  i = 0;
  while (i < 10) {
    num[i] = 0;
    i = i + 1;
  }
  return 0;
}

func readfloat(dummy) {
  dummy = cleannum(0);

  dot = 0;
  i = 0;
  x = 0;
  sign = 1;
  while (!end(f)) {
    c = read(f);
    if (c == '.') {
      assert(dot == 0);
      dot = 1;
    } else {
      if (c == '-') {
        assert(i == 0);
        sign = -1;
      } else {
        assert (c >= '0' && c <= '9');
        x = x * 10;
        x = x + (c - '0');
        assert(x >= 0);
      }
    }
    num[i] = c;
    i = i + 1;
  }
  return x * sign;
}

func readidx(n) {
  dummy = cleannum(0);

  idx = 0;
  i = 0;
  while (!end(f)) {
    c = read(f);
    assert(c >= '0' && c <= '9');
    idx = idx * 10;
    idx = idx + (c - '0');
    assert(idx >= 0);
    num[i] = c;
    i = i + 1;
  }

  assert(idx >= 1 && idx <= n);
  return idx;
}

main {
  n = 0;
  m = 0;
  inspectt (!end(f), f, '\n') {
    c = read(f);
    assert(c == 'v' || c == 'f');
    space = read(f);
    assert(space == ' ');
    if (c == 'v') {
      print('v');
      dim = 0;
      inspectt (1, f, ' ') {
        assert(!end(f));
        x = readfloat(0);
        print(' ');
        print(num);
        dim = dim + 1;
      }
      assert(dim == 3);
      print('\n');
      n = n + 1;
    } else {
      print('f');
      dim = 0;
      inspectt (1, f, ' ') {
        assert(!end(f));
        idx = readidx(n);
        print(' ');
        print(num);
        dim = dim + 1;
      }
      assert(dim >= 3);
      print('\n');
      m = m + 1;
    }
  }

  print(n);
  print('\n');
  print(m);
  print('\n');

  return 0;
}
```

C.2.2 Explicit Check Version

```
f = opent_ec("../inputs/obj/icosahedron-2.obj");
num = malloc_ec(10);
dim = 0;

func cleannum(dummy) {
  i = 0;
  while (i < 10) {
    num[i] = 0;
    i = i + 1;
  }
  return 0;
}

func readfloat(dummy) {
  dummy = cleannum(0);

  dot = 0;
  i = 0;
  x = 0;
  sign = 1;
  while (!end_ec(f)) {
    assert(i < 10);
    c = read_ec(f);
    if (c == '.') {
      assert(dot == 0);
      dot = 1;
    } else {
      if (c == '-') {
        assert(i == 0);
        sign = -1;
      } else {
        assert (c >= '0' && c <= '9');
        x = x * 10;
        x = x + (c - '0');
        assert(x >= 0);
      }
    }
    num[i] = c;
    i = i + 1;
  }
  return x * sign;
}

func readidx(n) {
  dummy = cleannum(0);

  idx = 0;
  i = 0;
  while (!end_ec(f)) {
    assert(i < 10);
    c = read_ec(f);
    assert(c >= '0' && c <= '9');
    idx = idx * 10;
    idx = idx + (c - '0');
    assert(idx >= 0);
    num[i] = c;
    i = i + 1;
  }

  assert(idx >= 1 && idx <= n);
  return idx;
}

main {
  assert(valid(f) && valid(num));
  n = 0;
  m = 0;
  inspectt (!end(f), f, '\n') {
    c = read_ec(f);
    assert(c == 'v' || c == 'f');
    space = read_ec(f);
    assert(space == ' ');
    if (c == 'v') {
      print('v');
      dim = 0;
      inspectt (1, f, ' ') {
        assert(!end(f));
        x = readfloat(0);
        print(' ');
        print(num);
        dim = dim + 1;
      }
      assert(dim == 3);
      print('\n');
      n = n + 1;
    } else {
      print('f');
      dim = 0;
      inspectt (1, f, ' ') {
        assert(!end(f));
        idx = readidx(n);
        print(' ');
        print(num);
        dim = dim + 1;
      }
      assert(dim >= 3);
      print('\n');
      m = m + 1;
    }
  }

  print(n);
  print('\n');
  print(m);
  print('\n');

  return 0;
}
```

C.2.3 Explicit Recovery Version

```
f = opent_ec("../inputs/obj/icosahedron-2.obj");
num = malloc_ec(100);
numidx = 0;
dim = 0;

bad = 0;

func cleannum(dummy) {
  i = 0;
  while (i < 100) {
    num[i] = 0;
    i = i + 1;
  }
  numidx = 0;
  return 0;
}

func putnum(c) {
  if (numidx >= 100) {
    bad = 1;
  } else {
    num[numidx] = c;
    numidx = numidx + 1;
  }
  return 0;
}

func revertnum(start) {
  while (numidx > start) {
    numidx = numidx - 1;
    num[numidx] = 0;
  }
  return 0;
}

func readfloat(dummy) {
  dummy = putnum(' ');

  dot = 0;
  i = 0;
  x = 0;
  sign = 1;
  while (!end_ec(f)) {
    c = read_ec(f);
    if (i >= 10 || numidx >= 100) {
      bad = 1;
    } else {
      if (c == '.') {
        if (dot != 0) {
```

```
            bad = 1;
          }
          dot = 1;
        } else {
          if (c == '-') {
            if (i != 0) {
              bad = 1;
            }
            sign = -1;
          } else {
            if (c >= '0' && c <= '9') {
              x = x * 10;
              x = x + (c - '0');
              if (x < 0) {
                bad = 1;
              }
            } else {
              bad = 1;
            }
          }
        }
        dummy = putnum(c);
        i = i + 1;
      }
    }
    return x * sign;
}

func readidx(n) {
    dummy = putnum(' ');

    idx = 0;
    i = 0;
    while (!end_ec(f)) {
      c = read_ec(f);
      if (i >= 10 || numidx >= 100) {
        bad = 1;
      } else {
        if (c >= '0' && c <= '9') {
          idx = idx * 10;
          idx = idx + (c - '0');
          if (idx < 0) {
            bad = 1;
          }
        } else {
          bad = 1;
        }
      }
      dummy = putnum(c);
      i = i + 1;
    }

    if (!(idx >= 1 && idx <= n)) {
      bad = 1;
    }
    return idx;
}

main {
    if (!(valid(f) && valid(num))) {
      exit(1);
    }
    n = 0;
    m = 0;
    lookatt (!end_ec(f), f, '\n') {
      c = read_ec(f);
      if (c == 'v' || c == 'f') {
        space = read_ec(f);
        if (space == ' ') {
          if (c == 'v') {
            dim = 0;
            dummy = cleannum(0);
            lookatt (1, f, ' ') {
              bad = 0;
              if (!end_ec(f)) {
                start = numidx;
                x = readfloat(0);
                if (!bad) {
                  dim = dim + 1;
                } else { // discard bad updates
                  dummy = revertnum(start);
                }
              }
            }
            if (dim == 3) {
              print('v');
              print(num);
              print('\n');
              n = n + 1;
            } // skip unit
          } else {
            dim = 0;
            dummy = cleannum(0);
            lookatt (1, f, ' ') {
              bad = 0;
              if (!end_ec(f)) {
                start = numidx;
                idx = readidx(n);
                if (!bad) {
                  dim = dim + 1;
                } else { // discard bad updates
                  dummy = revertnum(start);
                }
              }
            }
            if (dim >= 3) {
              print('f');
              print(num);
              print('\n');
              m = m + 1;
            } // skip unit
          }
        } // skip unit
      } // skip unit
    }

    print(n);
    print('\n');
    print(m);
    print('\n');

    return 0;
}
```

C.2.4 Conventional Version

```
f = opent_ec("../inputs/obj/icosahedron-2.obj");
num = malloc_ec(100);
numidx = 0;
dim = 0;

bad = 0;
endline = 0;

func cleannum(dummy) {
  i = 0;
  while (i < 100) {
    num[i] = 0;
    i = i + 1;
  }
  numidx = 0;
  return 0;
}

func putnum(c) {
  if (numidx >= 100) {
    bad = 1;
  } else {
    num[numidx] = c;
    numidx = numidx + 1;
  }
  return 0;
}

func revertnum(start) {
  while (numidx > start) {
    numidx = numidx - 1;
    num[numidx] = 0;
  }
  return 0;
}

```

```
func readfloat(dummy) {
  dummy = putnum(' ');

  dot = 0;
  i = 0;
  x = 0;
  sign = 1;
  c = read_ec(f);
  while (c >= 0 && c != '\n' && c != ' ') {
    if (i >= 10 || numidx >= 100) {
      bad = 1;
    } else {
      if (c == '.') {
        if (dot != 0) {
          bad = 1;
        }
        dot = 1;
      } else {
        if (c == '-') {
          if (i != 0) {
            bad = 1;
          }
          sign = -1;
        } else {
          if (c >= '0' && c <= '9') {
            x = x * 10;
            x = x + (c - '0');
            if (x < 0) {
              bad = 1;
            }
          } else {
            bad = 1;
          }
        }
      }
      dummy = putnum(c);
      i = i + 1;
    }
    c = read_ec(f);
  }

  if (c != ' ') {
    endline = 1;
  }
  if (i <= 0) {
    bad = 1;
  }
  return x * sign;
}

func readidx(n) {
  dummy = putnum(' ');

  idx = 0;
  i = 0;
  c = read_ec(f);
  while (c >= 0 && c != '\n' && c != ' ') {
    if (i >= 10 || numidx >= 100) {
      bad = 1;
    } else {
      if (c >= '0' && c <= '9') {
        idx = idx * 10;
        idx = idx + (c - '0');
        if (idx < 0) {
          bad = 1;
        }
      } else {
        bad = 1;
      }
    }
    dummy = putnum(c);
    i = i + 1;
    c = read_ec(f);
  }

  if (c != ' ') {
    endline = 1;
  }
  if (!(i > 0 && idx >= 1 && idx <= n)) {
    bad = 1;
  }
  return idx;
}

main {
  if (!(valid(f) && valid(num))) {
    exit(1);
  }
  n = 0;
  m = 0;
  while (!end_ec(f)) {
    c = read_ec(f);
    if (c == 'v' || c == 'f') {
      space = read_ec(f);
      if (space == ' ') {
        if (c == 'v') {
          dim = 0;
          dummy = cleannum(0);
          endline = 0;
          while (!endline) {
            bad = 0;
            start = numidx;
            x = readfloat(0);
            if (!bad) {
              dim = dim + 1;
            } else { // discard bad updates
              dummy = revertnum(start);
            }
          }
          if (dim == 3) {
            print('v');
            print(num);
            print('\n');
            n = n + 1;
          } // skip unit
        } else {
          dim = 0;
          dummy = cleannum(0);
          endline = 0;
          while (!endline) {
            bad = 0;
            start = numidx;
            idx = readidx(n);
            if (!bad) {
              dim = dim + 1;
            } else { // discard bad updates
              dummy = revertnum(start);
            }
          }
          if (dim >= 3) {
            print('f');
            print(num);
            print('\n');
            m = m + 1;
          } // skip unit
        }
        dummy = seek_ec(f, pos_ec(f) - 1);
        c = read_ec(f);
      } else { // skip unit
        c = space;
      }
    } // skip unit
    while (c >= 0 && c != '\n') {
      c = read_ec(f);
    }
  }

  print(n);
  print('\n');
  print(m);
  print('\n');

  return 0;
}
```

C.3 JSON

JSON files describe object attributes. A JSON file contains a unit which we define as follows. A token is a string that does not contain commas, colons, brackets, or braces. A key-value pair contains a token, a colon, and a unit. An object is

at least one key-value pair surrounded by braces. An array is at least one unit surrounded by brackets. A unit can be a token, an object or an array. Commas separate the key-value pairs inside objects and the units inside arrays.

The benchmark program reads a JSON file, expecting that there are at most 10 tokens, and that each token is at most 20 characters long. The program outputs the contents of inputs that do not trigger errors. In effect, the program discards tokens that are longer than 20 characters, malformed key-value pairs in objects, malformed units in arrays, and units that would otherwise make the program store more than 10 tokens.

C.3.1 Full RIFL Version

```
tokenstart = malloc(10);
tokenlen = malloc(10);
tokentype = malloc(10);
tokencount = 0;

func DFS(f, level) {
  first = read(f);
  // remove leading whitespace
  while (first == ' ' || first == '\n') {
    first = read(f);
  }
  assert (first != ',' && first != ':' && first != '}' && first != ']');
  if (first == '{') { // object
    objlen = 0;
    inspectt (1, f, ',', '}') {
      c = read(f);
      key = malloc(20);
      while (c == ' ' || c == '\n') {
        c = read(f);
      }
      // parse key token
      tokenstart[tokencount] = pos(f);
      i = 0;
      while (c != ':') {
        assert(c != '[' && c != '{');
        if (c == '\n') {
          key[i] = ' ';
        } else {
          key[i] = c;
        }
        i = i + 1;
        c = read(f);
      }
      // remove trailing whitespace
      while (i > 0 && (key[i-1] == ' ' || key[i-1] == '\n')) {
        key[i-1] = 0;
        i = i - 1;
      }
      // maintain global arrays
      tokenlen[tokencount] = i;
      tokentype[tokencount] = 'k';
      tokencount = tokencount + 1;
      // print key token
      j = 0;
      print('\n');
      while (j < level) {
        print(' ');
        print(' ');
        j = j + 1;
      }
      print(key);
      free(key);
      print(':');
      print(' ');
      // parse value
      vallen = DFS(f, level + 1);
      assert (vallen > 0);
      objlen = objlen + vallen + i;
    }
    return objlen;
  } else {
    if (first == '[') { // array
      arrlen = 0;
      print('\n');
      // parse array elements
      inspectt (1, f, ',', ']') {
        i = 0;
        while (i < level) {
          print(' ');
          print(' ');
          i = i + 1;
        }
        print('-');
        print(' ');
        elemlen = DFS(f, level + 1);
        assert (elemlen > 0);
        print('\n');
        arrlen = arrlen + elemlen;
      }
      return arrlen;
    } else { // single token
      word = malloc(20);
      // parse value token
      tokenstart[tokencount] = pos(f);
      word[0] = first;
      i = 1;
      while (!end(f)) {
        c = read(f);
        assert(c != ':');
        if (c == '\n') {
          word[i] = ' ';
        } else {
          word[i] = c;
        }
        i = i + 1;
      }
      // remove trailing whitespace
      while (i > 0 && (word[i-1] == ' ' || word[i-1] == '\n')) {
        word[i-1] = 0;
        i = i - 1;
      }
      // maintain global arrays
      tokenlen[tokencount] = i;
      tokentype[tokencount] = 'v';
      tokencount = tokencount + 1;
      // print value token
      print(word);
      free(word);
      return i;
    }
  }
}

main {
  f = opent("../inputs/json/widget.json");
  total = DFS(f, 0);
  print('\n');

  print(tokencount);
  print('\n');
  print(tokenstart);
  print('\n');
  print(tokenlen);
  print('\n');
  print(tokentype);
  print('\n');
  return total;
}
```

C.3.2 Explicit Check Version

```
tokenstart = malloc_ec(10);
tokenlen = malloc_ec(10);
tokentype = malloc_ec(10);
tokencount = 0;

func DFS(f, level) {
  first = read_ec(f);
```

```
  // remove leading whitespace
  while (first == ' ' || first == '\n') {
    first = read_ec(f);
  }
  assert (first >= 0 && first != ',' && first !=
       ':' && first != '}' && first != ']');
  if (first == '{') { // object
    objlen = 0;
    inspectt (tokencount < 9, f, ',', '}') {
      c = read_ec(f);
      key = malloc_ec(20);
      assert(valid(key));
      while (c == ' ' || c == '\n') {
        c = read_ec(f);
      }
      // parse key token
      tokenstart[tokencount] = pos_ec(f);
      assert(tokenstart[tokencount] >= 0);
      i = 0;
      while (c != ':') {
        assert(i < 20 && c >= 0 && c != '[' && c
             != '{');
        if (c == '\n') {
          key[i] = ' ';
        } else {
          key[i] = c;
        }
        i = i + 1;
        c = read_ec(f);
      }
      // remove trailing whitespace
      while (i > 0 && (key[i-1] == ' ' || key[i-1]
           == '\n')) {
        key[i-1] = 0;
        i = i - 1;
      }
      // maintain global arrays
      tokenlen[tokencount] = i;
      tokentype[tokencount] = 'k';
      tokencount = tokencount + 1;
      // print key token
      j = 0;
      print('\n');
      while (j < level) {
        print(' ');
        print(' ');
        j = j + 1;
      }
      print(key);
      dummy = free_ec(key);
      print(':');
      print(' ');
      // parse value
      vallen = DFS(f, level + 1);
      assert (vallen > 0);
      objlen = objlen + vallen + i;
    }
    return objlen;
  } else {
    if (first == '[') { // array
      arrlen = 0;
      print('\n');
      // parse array elements
      inspectt (1, f, ',', ']') {
        i = 0;
        while (i < level) {
          print(' ');
          print(' ');
          i = i + 1;
        }
        print('-');
        print(' ');
        elemlen = DFS(f, level + 1);
        assert (elemlen > 0);
        print('\n');
        arrlen = arrlen + elemlen;
      }
      return arrlen;
    } else { // single token
      word = malloc_ec(20);
      assert(valid(word));
      // parse value token
      tokenstart[tokencount] = pos_ec(f);
      word[0] = first;
      i = 1;
      while (!end_ec(f)) {
        assert(i < 20);
        c = read_ec(f);
        assert(c >= 0 && c != ':');
        if (c == '\n') {
          word[i] = ' ';
        } else {
          word[i] = c;
        }
        i = i + 1;
      }
      // remove trailing whitespace
      while (i > 0 && (word[i-1] == ' ' || word[i
           -1] == '\n')) {
        word[i-1] = 0;
        i = i - 1;
      }
      // maintain global arrays
      tokenlen[tokencount] = i;
      tokentype[tokencount] = 'v';
      tokencount = tokencount + 1;
      // print value token
      print(word);
      dummy = free_ec(word);
      return i;
    }
  }
}

main {
  f = opent_ec("../inputs/json/widget.json");
  assert(valid(f) && valid(tokenstart) && valid(
       tokenlen) && valid(tokentype));
  total = DFS(f, 0);
  print('\n');

  print(tokencount);
  print('\n');
  print(tokenstart);
  print('\n');
  print(tokenlen);
  print('\n');
  print(tokentype);
  print('\n');
  return total;
}
```

C.3.3 Explicit Recovery Version

```
tokenstart = malloc_ec(10);
tokenlen = malloc_ec(10);
tokentype = malloc_ec(10);
tokencount = 0;

stack = malloc_ec(150);
stackidx = 0;
stackbad = 0;

func putstack(c) {
  if (stackidx >= 150) {
    stackbad = 1;
  } else {
    stack[stackidx] = c;
    stackidx = stackidx + 1;
  }
  return 0;
}

func revertstack(start) {
  while (stackidx > start) {
    if (stackidx < 150) {
      stack[stackidx] = 0;
    }
    stackidx = stackidx - 1;
  }
  if (stackidx < 150) {
    stackbad = 0;
```

```
  }
  return 0;
}

func reverttoken(start) {
  while (tokencount > start) {
    if (tokencount < 10) {
      tokenstart[tokencount] = 0;
      tokenlen[tokencount] = 0;
      tokentype[tokencount] = 0;
    }
    tokencount = tokencount - 1;
  }
  return 0;
}

func DFS(f, level) {
  first = read_ec(f);
  // remove leading whitespace
  while (first == ' ' || first == '\n') {
    first = read_ec(f);
  }
  if (!(first >= 0 && first != ',' && first != ':'
       && first != '}' && first != ']')) {
    return -1;
  }
  if (first == '{') { // object
    objlen = 0;
    lookatt (tokencount < 9, f, ',', '}') {
      bad = 0;
      c = read_ec(f);
      key = malloc_ec(20);
      if (valid(key)) {
        while (c == ' ' || c == '\n') {
          c = read_ec(f);
        }
        // parse key token
        position = pos_ec(f);
        if (position < 0) {
          bad = 1;
        }
        i = 0;
        while (!bad && c != ':') {
          if (!(i < 20 && c >= 0 && c != '[' && c
               != '{')) {
            bad = 1;
          }
          if (c == '\n') {
            key[i] = ' ';
          } else {
            key[i] = c;
          }
          i = i + 1;
          c = read_ec(f);
        }
        if (!bad) {
          stackok = stackidx;
          tokenok = tokencount;
          // remove trailing whitespace
          while (i > 0 && (key[i-1] == ' ' || key[
               i-1] == '\n')) {
            key[i-1] = 0;
            i = i - 1;
          }
          // maintain global arrays
          tokenstart[tokencount] = position;
          tokenlen[tokencount] = i;
          tokentype[tokencount] = 'k';
          tokencount = tokencount + 1;
          // print key token
          j = 0;
          dummy = putstack('\n');
          while (j < level) {
            dummy = putstack(' ');
            dummy = putstack(' ');
            j = j + 1;
          }
          j = 0;
          while (j < i) {
            x = key[j];
            dummy = putstack(x);
            j = j + 1;
          }
          dummy = putstack(':');
          dummy = putstack(' ');
          // parse value
          vallen = DFS(f, level + 1);
          if (!stackbad && vallen > 0) {
            objlen = objlen + vallen + i;
          } else { // discard bad updates
            dummy = revertstack(stackok);
            dummy = reverttoken(tokenok);
          }
        } // skip unit
        dummy = free_ec(key);
      } // skip unit
    }
    return objlen;
  } else {
    if (first == '[') { // array
      arrlen = 0;
      dummy = putstack('\n');
      // parse array elements
      lookatt (1, f, ',', ']') {
        stackok = stackidx;
        tokenok = tokencount;
        i = 0;
        while (i < level) {
          dummy = putstack(' ');
          dummy = putstack(' ');
          i = i + 1;
        }
        dummy = putstack('-');
        dummy = putstack(' ');
        elemlen = DFS(f, level + 1);
        dummy = putstack('\n');
        if (!stackbad && elemlen > 0) {
          arrlen = arrlen + elemlen;
        } else { // discard bad updates
          dummy = revertstack(stackok);
          dummy = reverttoken(tokenok);
        }
      }
      return arrlen;
    } else { // single token
      word = malloc_ec(20);
      bad = 0;
      if (valid(word)) {
        // parse value token
        position = pos_ec(f);
        if (position < 0) {
          bad = 1;
        }
        word[0] = first;
        i = 1;
        while (!bad && !end_ec(f)) {
          if (i >= 20) {
            bad = 1;
          } else {
            c = read_ec(f);
            if (!(c >= 0 && c != ':')) {
              bad = 1;
            } else {
              if (c == '\n') {
                word[i] = ' ';
              } else {
                word[i] = c;
              }
            }
            i = i + 1;
          }
        }
        if (!bad) {
          stackok = stackidx;
          // remove trailing whitespace
          while (i > 0 && (word[i-1] == ' ' ||
               word[i-1] == '\n')) {
            word[i-1] = 0;
            i = i - 1;
          }
          // maintain global arrays
          tokenstart[tokencount] = position;
```

```
            tokenlen[tokencount] = i;
            tokentype[tokencount] = 'v';
            tokencount = tokencount + 1;
            // print value token
            j = 0;
            while (j < i) {
              x = word[j];
              dummy = putstack(x);
              j = j + 1;
            }
            if (stackbad) { // discard bad updates
              dummy = revertstack(stackok);
              dummy = reverttoken(tokencount - 1);
              i = 0;
            }
          } else { // skip unit
            i = 0;
          }
          dummy = free_ec(word);
          return i;
        } else { // skip unit
          return -1;
        }
      }
    }
  }

  main {
    f = opent_ec("../inputs/json/widget.json");
    if (!(valid(f) && valid(tokenstart) && valid(
         tokenlen) && valid(tokentype) && valid(
         stack))) {
      exit(1);
    }
    total = DFS(f, 0);
    if (total < 0) {
      exit(1);
    } else {
      print(stack);
      print('\n');

      print(tokencount);
      print('\n');
      print(tokenstart);
      print('\n');
      print(tokenlen);
      print('\n');
      print(tokentype);
      print('\n');
    }
    return total;
  }
```

C.3.4 Conventional Version

```
  tokenstart = malloc_ec(10);
  tokenlen = malloc_ec(10);
  tokentype = malloc_ec(10);
  tokencount = 0;

  stack = malloc_ec(150);
  stackidx = 0;
  stackbad = 0;

  func putstack(c) {
    if (stackidx >= 150) {
      stackbad = 1;
    } else {
      stack[stackidx] = c;
      stackidx = stackidx + 1;
    }
    return 0;
  }

  func revertstack(start) {
    while (stackidx > start) {
      if (stackidx < 150) {
        stack[stackidx] = 0;
      }
      stackidx = stackidx - 1;
    }
    if (stackidx < 150) {
      stackbad = 0;
    }
    return 0;
  }

  func reverttoken(start) {
    while (tokencount > start) {
      if (tokencount < 10) {
        tokenstart[tokencount] = 0;
        tokenlen[tokencount] = 0;
        tokentype[tokencount] = 0;
      }
      tokencount = tokencount - 1;
    }
    return 0;
  }

  func DFS(f, level) {
    first = read_ec(f);
    // remove leading whitespace
    while (first == ' ' || first == '\n') {
      first = read_ec(f);
    }
    if (!(first >= 0 && first != ',' && first != ':'
         && first != '}' && first != ']')) {
      return -1;
    }
    if (first == '{') { // object
      objlen = 0;
      finish = 0;
      while (tokencount < 9 && !finish) {
        bad = 0;
        c = read_ec(f);
        key = malloc_ec(20);
        if (valid(key)) {
          while (c == ' ' || c == '\n') {
            c = read_ec(f);
          }
          // parse key token
          position = pos_ec(f);
          if (position < 0) {
            bad = 1;
          }
          i = 0;
          while (!bad && c != ':' && c != ',' && c
               != '}' && c != ']') {
            if (!(i < 20 && c >= 0 && c != '[' && c
                 != '{')) {
              bad = 1;
            }
            if (c == '\n') {
              key[i] = ' ';
            } else {
              key[i] = c;
            }
            i = i + 1;
            c = read_ec(f);
          }
          if (!bad) {
            stackok = stackidx;
            tokenok = tokencount;
            // remove trailing whitespace
            while (i > 0 && (key[i-1] == ' ' || key[
                 i-1] == '\n')) {
              key[i-1] = 0;
              i = i - 1;
            }
            // maintain global arrays
            tokenstart[tokencount] = position;
            tokenlen[tokencount] = i;
            tokentype[tokencount] = 'k';
            tokencount = tokencount + 1;
            // print key token
            j = 0;
            dummy = putstack('\n');
            while (j < level) {
              dummy = putstack(' ');
              dummy = putstack(' ');
              j = j + 1;
            }
```

```
104           j = 0;
105           while (j < i) {
106             x = key[j];
107             dummy = putstack(x);
108             j = j + 1;
109           }
110           dummy = putstack(':');
111           dummy = putstack(' ');
112           // parse value
113           vallen = DFS(f, level + 1);
114           if (!stackbad && vallen > 0) {
115             objlen = objlen + vallen + i;
116           } else { // discard bad updates
117             dummy = revertstack(stackok);
118             dummy = reverttoken(tokenok);
119           }
120         } // skip unit
121         dummy = free_ec(key);
122       } // skip unit
123       while (c >= 0 && c != ',' && c != '}' && c
              != ']') { // end of
              unit
124         c = read_ec(f);
125       }
126       if (c < 0 || c == '}') { // end of sequence
127         finish = 1;
128       }
129       if (c == ']') { // outside delimiters
130         finish = 1;
131         dummy = seek_ec(f, pos_ec(f) - 1);
132       }
133     }
134     return objlen;
135   } else {
136     if (first == '[') { // array
137       arrlen = 0;
138       dummy = putstack('\n');
139       // parse array elements
140       while (!finish) {
141         stackok = stackidx;
142         tokenok = tokencount;
143         i = 0;
144         while (i < level) {
145           dummy = putstack(' ');
146           dummy = putstack(' ');
147           i = i + 1;
148         }
149         dummy = putstack('-');
150         dummy = putstack(' ');
151         elemlen = DFS(f, level + 1);
152         dummy = putstack('\n');
153         if (!stackbad && elemlen > 0) {
154           arrlen = arrlen + elemlen;
155         } else { // discard bad updates
156           dummy = revertstack(stackok);
157           dummy = reverttoken(tokenok);
158         }
159         c = read_ec(f);
160         while (c >= 0 && c != ',' && c != ']' && c
                != '}') { // end of
                unit
161           c = read_ec(f);
162         }
163         if (c < 0 || c == ']') { // end of sequence
164           finish = 1;
165         }
166         if (c == '}') { // outside delimiters
167           finish = 1;
168           dummy = seek_ec(f, pos_ec(f) - 1);
169         }
170       }
171       return arrlen;
172     } else { // single token
173       word = malloc_ec(20);
174       bad = 0;
175       if (valid(word)) {
176         // parse value token
177         position = pos_ec(f);
178         if (position < 0) {
179           bad = 1;
180         }
181         word[0] = first;
182         i = 1;
183         c = read_ec(f);
184         while (!bad && c >= 0 && c != ',' && c !=
                ']' && c != '}') {
185           if (i >= 20) {
186             bad = 1;
187           } else {
188             if (!(c >= 0 && c != ':')) {
189               bad = 1;
190             } else {
191               if (c == '\n') {
192                 word[i] = ' ';
193               } else {
194                 word[i] = c;
195               }
196             }
197             i = i + 1;
198             c = read_ec(f);
199           }
200         }
201         if (c == ',' || c == ']' || c == '}') {
202           dummy = seek_ec(f, pos_ec(f) - 1);
203         }
204         if (!bad) {
205           stackok = stackidx;
206           // remove trailing whitespace
207           while (i > 0 && (word[i-1] == ' ' ||
                word[i-1] == '\n')) {
208             word[i-1] = 0;
209             i = i - 1;
210           }
211           // maintain global arrays
212           tokenstart[tokencount] = position;
213           tokenlen[tokencount] = i;
214           tokentype[tokencount] = 'v';
215           tokencount = tokencount + 1;
216           // print value token
217           j = 0;
218           while (j < i) {
219             x = word[j];
220             dummy = putstack(x);
221             j = j + 1;
222           }
223           if (stackbad) { // discard bad updates
224             dummy = revertstack(stackok);
225             dummy = reverttoken(tokencount - 1);
226             i = 0;
227           }
228         } else { // skip unit
229           i = 0;
230         }
231         dummy = free_ec(word);
232         return i;
233       } else { // skip unit
234         return -1;
235       }
236     }
237   }
238 }
239
240 main {
241   f = opent_ec("../inputs/json/widget.json");
242   if (!(valid(f) && valid(tokenstart) && valid(
          tokenlen) && valid(tokentype) && valid(
          stack))) {
243     exit(1);
244   }
245   total = DFS(f, 0);
246   if (total < 0) {
247     exit(1);
248   } else {
249     print(stack);
250     print('\n');
251
252     print(tokencount);
253     print('\n');
254     print(tokenstart);
255     print('\n');
256     print(tokenlen);
257     print('\n');
```

```
258       print(tokentype);
259       print('\n');
260     }
261     return total;
262   }
```

C.4 PNG

PNG files store images. A PNG file contains a magic string followed by chunks. Each chunk contains a 4-byte nonnegative length field, a 4-byte type field, "length" bytes of data, and a 4-byte cyclic redundancy code (CRC).

The benchmark program reads a PNG file, expecting that each chunk is small enough to fit in the heap memory. When a chunk length is too large, the program flushes the input until the end of the chunk and keeps parsing. The program outputs the data contents of all chunks with type "IDAT" that do not trigger errors. The program rejects PNG files with malformed headers. The program discards trailing inputs after seeing a chunk with a negative length.

C.4.1 Full RIFL Version

```
1   func readintbytes (f, n) {
2     x = 0;
3     i = 0;
4     while (i < n) {
5       byte = read(f);
6       x = x << 8;
7       x = x | byte;
8       i = i + 1;
9     }
10    return x;
11  }
12
13  main {
14    f = openb("../inputs/png0/oi4n0g16-2.png", 0);
15
16    magic = malloc(8);
17    i = 0;
18    while (i < 8) {
19      x = read(f);
20      magic[i] = x;
21      i = i + 1;
22    }
23    assert(magic[0] == 137 && magic[1] == 'P' &&
          magic[2] == 'N' && magic[3] == 'G' && magic
          [4] == 13 && magic[5] == 10 && magic[6] ==
          26 && magic[7] == 10);
24    free(magic);
25
26    n = readintbytes(f, 4);
27    assert(n == 13);
28
29    ihdr = malloc(4);
30    i = 0;
31    while (i < 4) {
32      x = read(f);
33      ihdr[i] = x;
34      i = i + 1;
35    }
36    assert(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr
          [2] == 'D' && ihdr[3] == 'R');
37    free(ihdr);
38
39    w = readintbytes(f, 4);
40    h = readintbytes(f, 4);
41    assert(w > 0 && h > 0);
42
43    depth = read(f);
44    color = read(f);
45    compression = read(f);
46    filter = read(f);
47    interlace = read(f);
48    assert((depth == 1 || depth == 2 || depth == 4
          || depth == 8 || depth == 16) && color == 0
           && compression == 0 && filter == 0 && (
          interlace == 0 || interlace == 1));
49
50    crc = readintbytes(f, 4);
51    // check CRC
52
53    ndata = 0;
54    finish = 0;
55    type = malloc(4);
56    inspectb (!finish, f, 0, 4, 8) {
57      length = readintbytes(f, 4);
58
59      i = 0;
60      while (i < 4) {
61        x = read(f);
62        type[i] = x;
63        i = i + 1;
64      }
65      // check type legal
66
67      if (type[0] == 'I' && type[1] == 'E' && type
            [2] == 'N' && type[3] == 'D') { //
            IEND
68        a = read(f);
69        b = read(f);
70        c = read(f);
71        d = read(f);
72        assert(length == 0 && a == 174 && b == 66 &&
              c == 96 && d == 130);
73        finish = 1;
74      } else {
75        bytes = malloc(length);
76        i = 0;
77        while (i < length) {
78          x = read(f);
79          bytes[i] = x;
80          i = i + 1;
81        }
82        crc = readintbytes(f, 4);
83        // check CRC
84        if (type[0] == 'I' && type[1] == 'D' && type
              [2] == 'A' && type[3] == 'T') { //
              IDAT
85          ndata = ndata + 1;
86          // decompress
87          // decode
88          print(bytes);
89        }
90        free(bytes);
91      }
92    }
93    free(type);
94    return 0;
95  }
```

C.4.2 Explicit Check Version

```
1   func readintbytes (f, n) {
2     x = 0;
3     i = 0;
4     while (i < n) {
5       byte = read_ec(f);
6       assert(byte >= 0);
7       x = x << 8;
8       x = x | byte;
9       i = i + 1;
10    }
11    return x;
12  }
13
14  main {
15    f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0)
          ;
16
17    magic = malloc_ec(8);
18    assert(valid(f) && valid(magic));
19    i = 0;
20    while (i < 8) {
21      x = read_ec(f);
```

```
22      assert(x >= 0);
23      magic[i] = x;
24      i = i + 1;
25    }
26    assert(magic[0] == 137 && magic[1] == 'P' &&
          magic[2] == 'N' && magic[3] == 'G' && magic
          [4] == 13 && magic[5] == 10 && magic[6] ==
          26 && magic[7] == 10);
27    x = free_ec(magic);
28
29    n = readintbytes(f, 4);
30    assert(n == 13);
31
32    ihdr = malloc_ec(4);
33    assert(valid(ihdr));
34    i = 0;
35    while (i < 4) {
36      x = read_ec(f);
37      assert(x >= 0);
38      ihdr[i] = x;
39      i = i + 1;
40    }
41    assert(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr
          [2] == 'D' && ihdr[3] == 'R');
42    x = free_ec(ihdr);
43
44    w = readintbytes(f, 4);
45    h = readintbytes(f, 4);
46    assert(w > 0 && h > 0);
47
48    depth = read_ec(f);
49    color = read_ec(f);
50    compression = read_ec(f);
51    filter = read_ec(f);
52    interlace = read_ec(f);
53    assert((depth == 1 || depth == 2 || depth == 4
          || depth == 8 || depth == 16) && color == 0
           && compression == 0 && filter == 0 && (
          interlace == 0 || interlace == 1));
54
55    crc = readintbytes(f, 4);
56    // check CRC
57
58    ndata = 0;
59    finish = 0;
60    type = malloc_ec(4);
61    assert(valid(type));
62    inspectb (!finish, f, 0, 4, 8) {
63      length = readintbytes(f, 4);
64
65      i = 0;
66      while (i < 4) {
67        x = read_ec(f);
68        assert(x >= 0);
69        type[i] = x;
70        i = i + 1;
71      }
72      // check type legal
73
74      if (type[0] == 'I' && type[1] == 'E' && type
            [2] == 'N' && type[3] == 'D') { //
            IEND
75        a = read_ec(f);
76        b = read_ec(f);
77        c = read_ec(f);
78        d = read_ec(f);
79        assert(length == 0 && a == 174 && b == 66 &&
              c == 96 && d == 130);
80        finish = 1;
81      } else {
82        assert(length > 0);
83        bytes = malloc_ec(length);
84        assert(valid(bytes));
85        i = 0;
86        while (i < length) {
87          x = read_ec(f);
88          assert(x >= 0);
89          bytes[i] = x;
90          i = i + 1;
91        }
92        crc = readintbytes(f, 4);
93        // check CRC
94        if (type[0] == 'I' && type[1] == 'D' && type
              [2] == 'A' && type[3] == 'T') { //
              IDAT
95          ndata = ndata + 1;
96          // decompress
97          // decode
98          print(bytes);
99        }
100       x = free_ec(bytes);
101     }
102   }
103   x = free_ec(type);
104   return 0;
105 }
```

C.4.3 Explicit Recovery Version

```
1   bad = 0;
2
3   func readintbytes (f, n) {
4     x = 0;
5     i = 0;
6     while (i < n && !bad) {
7       byte = read_ec(f);
8       if (byte >= 0) {
9         x = x << 8;
10        x = x | byte;
11        i = i + 1;
12      } else {
13        bad = 1;
14        return -1;
15      }
16    }
17    return x;
18  }
19
20  main {
21    f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0)
          ;
22
23    magic = malloc_ec(8);
24    if (!valid(f) || !valid(magic)) {
25      exit(1);
26    }
27    i = 0;
28    while (i < 8) {
29      x = read_ec(f);
30      if (x < 0) {
31        exit(1);
32      }
33      magic[i] = x;
34      i = i + 1;
35    }
36    if (!(magic[0] == 137 && magic[1] == 'P' &&
          magic[2] == 'N' && magic[3] == 'G' && magic
          [4] == 13 && magic[5] == 10 && magic[6] ==
          26 && magic[7] == 10)) {
37      exit(1);
38    }
39    x = free_ec(magic);
40
41    n = readintbytes(f, 4);
42    if (bad || n != 13) {
43      exit(1);
44    }
45
46    ihdr = malloc_ec(4);
47    if (!valid(ihdr)) {
48      exit(1);
49    }
50    i = 0;
51    while (i < 4) {
52      x = read_ec(f);
53      if (x < 0) {
54        exit(1);
55      }
56      ihdr[i] = x;
57      i = i + 1;
58    }
```

```
  if (!(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr
       [2] == 'D' && ihdr[3] == 'R')) {
    exit(1);
  }
  x = free_ec(ihdr);

  w = readintbytes(f, 4);
  h = readintbytes(f, 4);
  if (bad || w <= 0 || h <= 0) {
    exit(1);
  }

  depth = read_ec(f);
  color = read_ec(f);
  compression = read_ec(f);
  filter = read_ec(f);
  interlace = read_ec(f);
  if (!(depth == 1 || depth == 2 || depth == 4 ||
       depth == 8 || depth == 16) || color != 0 ||
        compression != 0 || filter != 0 || !(
       interlace == 0 || interlace == 1)) {
    exit(1);
  }

  crc = readintbytes(f, 4);
  if (bad) {
    exit(1);
  }
  // check CRC

  ndata = 0;
  finish = 0;
  type = malloc_ec(4);
  if (!valid(type)) {
    exit(1);
  }
  lookatb (!finish, f, 0, 4, 8) {
    bad = 0;
    length = readintbytes(f, 4);

    i = 0;
    while (i < 4) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
      }
      type[i] = x;
      i = i + 1;
    }
    // check type legal

    if (type[0] == 'I' && type[1] == 'E' && type
         [2] == 'N' && type[3] == 'D') { //
         IEND
      a = read_ec(f);
      b = read_ec(f);
      c = read_ec(f);
      d = read_ec(f);
      if (length == 0 && a == 174 && b == 66 && c
           == 96 && d == 130) { //
           good
        finish = 1;
      } // skip unit
    } else {
      if (length > 0) { // good
        bytes = malloc_ec(length);
        if (valid(bytes)) {
          i = 0;
          while (i < length) {
            x = read_ec(f);
            if (x < 0) {
              bad = 1;
            }
            bytes[i] = x;
            i = i + 1;
          }
          crc = readintbytes(f, 4);
          // check CRC
          if (!bad && type[0] == 'I' && type[1] ==
                'D' && type[2] == 'A' && type[3]
                == 'T') { //
                IDAT
            ndata = ndata + 1;
            // decompress
            // decode
            print(bytes);
          } // skip unit
          x = free_ec(bytes);
        } // skip unit
      } // skip unit
    }
  }
  x = free_ec(type);
  return 0;
}
```

C.4.4 Conventional Version

```
bad = 0;

func readintbytes (f, n) {
  x = 0;
  i = 0;
  while (i < n && !bad) {
    byte = read_ec(f);
    if (byte >= 0) {
      x = x << 8;
      x = x | byte;
      i = i + 1;
    } else {
      bad = 1;
      return -1;
    }
  }
  return x;
}

main {
  f = openb_ec("../inputs/png0/oi4n0g16-2.png", 0)
       ;

  magic = malloc_ec(8);
  if (!valid(f) || !valid(magic)) {
    exit(1);
  }
  i = 0;
  while (i < 8) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    magic[i] = x;
    i = i + 1;
  }
  if (!(magic[0] == 137 && magic[1] == 'P' &&
       magic[2] == 'N' && magic[3] == 'G' && magic
       [4] == 13 && magic[5] == 10 && magic[6] ==
       26 && magic[7] == 10)) {
    exit(1);
  }
  x = free_ec(magic);

  n = readintbytes(f, 4);
  if (bad || n != 13) {
    exit(1);
  }

  ihdr = malloc_ec(4);
  if (!valid(ihdr)) {
    exit(1);
  }
  i = 0;
  while (i < 4) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    ihdr[i] = x;
    i = i + 1;
  }
```

```
59    if (!(ihdr[0] == 'I' && ihdr[1] == 'H' && ihdr
          [2] == 'D' && ihdr[3] == 'R')) {
60      exit(1);
61    }
62    x = free_ec(ihdr);
63
64    w = readintbytes(f, 4);
65    h = readintbytes(f, 4);
66    if (bad || w <= 0 || h <= 0) {
67      exit(1);
68    }
69
70    depth = read_ec(f);
71    color = read_ec(f);
72    compression = read_ec(f);
73    filter = read_ec(f);
74    interlace = read_ec(f);
75    if (!(depth == 1 || depth == 2 || depth == 4 ||
          depth == 8 || depth == 16) || color != 0 ||
           compression != 0 || filter != 0 || !(
          interlace == 0 || interlace == 1)) {
76      exit(1);
77    }
78
79    crc = readintbytes(f, 4);
80    if (bad) {
81      exit(1);
82    }
83    // check CRC
84
85    ndata = 0;
86    finish = 0;
87    type = malloc_ec(4);
88    if (!valid(type)) {
89      exit(1);
90    }
91    while (!finish) {
92      bad = 0;
93      length = readintbytes(f, 4);
94      eou = pos_ec(f) + length + 8;
95      if (!bad && length >= 0) {
96        i = 0;
97        while (i < 4) {
98          x = read_ec(f);
99          if (x < 0) {
100           bad = 1;
101         }
102         type[i] = x;
103         i = i + 1;
104       }
105       // check type legal
106
107       if (type[0] == 'I' && type[1] == 'E' && type
              [2] == 'N' && type[3] == 'D') { //
              IEND
108         a = read_ec(f);
109         b = read_ec(f);
110         c = read_ec(f);
111         d = read_ec(f);
112         if (length == 0 && a == 174 && b == 66 &&
                c == 96 && d == 130) { //
                good
113           finish = 1;
114         } // skip unit
115       } else { // other types
116         if (length > 0) { // good
117           bytes = malloc_ec(length);
118           if (valid(bytes)) {
119             i = 0;
120             while (i < length) {
121               x = read_ec(f);
122               if (x < 0) {
123                 bad = 1;
124               }
125               bytes[i] = x;
126               i = i + 1;
127             }
128             crc = readintbytes(f, 4);
129             // check CRC
130             if (!bad && type[0] == 'I' && type[1]
                    == 'D' && type[2] == 'A' && type
                    [3] == 'T') { //
                    IDAT
131               ndata = ndata + 1;
132               // decompress
133               // decode
134               print(bytes);
135             } // skip unit
136             x = free_ec(bytes);
137           } // skip unit
138         } // skip unit
139       }
140       // go to next unit according to length
141       x = seek_ec(f, eou);
142       if (x < 0) { // give up
143         finish = 1;
144       }
145     } else { // give up
146       finish = 1;
147     }
148   }
149   x = free_ec(type);
150   return 0;
151 }
```

C.5 ZIP

ZIP files archive user files. A ZIP file contains a portion of archive data, a central directory, and an end of central directory (EOCD) record. The archive portion contains the archived user files. The central directory contains a list of records. Each record in the central directory describes metadata of a user file and the starting offset of this user file in the archive portion. The EOCD record is at the end of the ZIP file and describes the starting offset of the central directory.

The benchmark program reads a ZIP file, iterating over the records in the central directory. The program expects that each file name and the contents for each archived user file are both short enough to fit in the heap memory. When a file name is too long, the program flushes the input until the end of the record and keeps parsing. When the contents of an archived user file is too large, the program stops parsing the current record and continues with the next record. The program outputs (a) the file names as listed the central directory and (b) the file names and the contents in archived user files. The program rejects ZIP files with malformed EOCD records. The program discards trailing inputs after seeing a record with a negative file-name length.

C.5.1 Full RIFL Version

```
1   func readintbytesl (f, n) { // little endian
2     x = read(f);
3     if (n == 1) {
4       return x;
5     } else {
6       y = readintbytesl(f, n-1);
7       return (y << 8) + x;
8     }
9   }
10
11  main {
12    fdir = openb("../inputs/zip/stuff-1.zip", 1);
13    fent = openb("../inputs/zip/stuff-1.zip", 1);
14
15    eocd = size(fdir) - 4;
16    found = 0;
17    while (!found) {
18      seek(fdir, eocd);
19      dir_sig = readintbytesl(fdir, 4);
20      if (dir_sig == 101010256) { // 0x06054b50
```

```
      found = 1;
    } else {
      eocd = eocd - 1;
    }
  }
  assert(found);
  diskid = readintbytesl(fdir, 2);
  ndisk = readintbytesl(fdir, 2);
  dirid = readintbytesl(fdir, 2);
  ndir = readintbytesl(fdir, 2);

  dir_size = readintbytesl(fdir, 4);
  dir_start = readintbytesl(fdir, 4);
  eocd_comm_size = readintbytesl(fdir, 2);
  if (eocd_comm_size > 0) {
    eocd_comm = malloc(eocd_comm_size);
    i = 0;
    while (i < eocd_comm_size) {
      b = read(fdir);
      eocd_comm[i] = b;
      i = i + 1;
    }
  }

  seek(fdir, dir_start);
  inspectb (pos(fdir) < eocd, fdir, 28, 2, 40) {
    dir_sig = readintbytesl(fdir, 4);
    dir_ver_made = readintbytesl(fdir, 2);
    dir_ver_extr = readintbytesl(fdir, 2);
    dir_flag = readintbytesl(fdir, 2);
    dir_comp = readintbytesl(fdir, 2);
    dir_modif = readintbytesl(fdir, 4);
    dir_crc = readintbytesl(fdir, 4);
    dir_ent_size = readintbytesl(fdir, 4);
    dir_ent_size_uncomp = readintbytesl(fdir, 4);
    dir_name_size = readintbytesl(fdir, 2);
    dir_extr_size = readintbytesl(fdir, 2);
    dir_comm_size = readintbytesl(fdir, 2);
    dir_diskid = readintbytesl(fdir, 2);
    dir_attr = malloc(6);
    i = 0;
    while (i < 6) {
      b = read(fdir);
      dir_attr[i] = b;
      i = i + 1;
    }
    dir_ent_start = readintbytesl(fdir, 4);
    assert(dir_sig == 33639248 && dir_extr_size ==
          24 && dir_comm_size == 0);
    dir_name = malloc(dir_name_size);
    i = 0;
    while (i < dir_name_size) {
      b = read(fdir);
      dir_name[i] = b;
      i = i + 1;
    }
    print(dir_name);
    print('\n');
    dir_extr = malloc(dir_extr_size);
    i = 0;
    while (i < dir_extr_size) {
      b = read(fdir);
      dir_extr[i] = b;
      i = i + 1;
    }

    seek(fent, dir_ent_start);

    ent_sig = readintbytesl(fent, 4);
    ent_ver_extr = readintbytesl(fent, 2);
    ent_flag = readintbytesl(fent, 2);
    ent_comp = readintbytesl(fent, 2);
    ent_modif = readintbytesl(fent, 4);
    ent_crc = readintbytesl(fent, 4);
    ent_ent_size = readintbytesl(fent, 4);
    ent_ent_size_uncomp = readintbytesl(fent, 4);
    ent_name_size = readintbytesl(fent, 2);
    ent_extr_size = readintbytesl(fent, 2);

    assert(ent_sig == 67324752 && ent_ver_extr ==
         dir_ver_extr && ent_flag == dir_flag &&
         ent_comp == dir_comp && ent_modif ==
         dir_modif && ent_name_size ==
         dir_name_size);
    ent_name = malloc(ent_name_size);
    i = 0;
    while (i < ent_name_size) {
      b = read(fent);
      ent_name[i] = b;
      i = i + 1;
    }
    print(ent_name);
    print('\n');
    if (ent_extr_size > 0) {
      ent_extr = malloc(ent_extr_size);
      i = 0;
      while (i < ent_extr_size) {
        b = read(fent);
        ent_extr[i] = b;
        i = i + 1;
      }
    }
    assert(ent_flag & 8 == 0 && ent_crc == dir_crc
         && ent_ent_size == dir_ent_size &&
         ent_ent_size_uncomp ==
         dir_ent_size_uncomp);

    raw_data = malloc(ent_ent_size);
    i = 0;
    while (i < ent_ent_size) {
      b = read(fent);
      raw_data[i] = b;
      i = i + 1;
    }
    // decompress
    // check crc
    print(raw_data);
    print('\n');
    free(ent_name);
    free(raw_data);
  }
  seek(fdir, eocd);
  return 0;
}
```

C.5.2 Explicit Check Version

```
func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  assert(x >= 0);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

main {
  fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
  fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
  assert(valid(fdir) && valid(fent));

  eocd = size_ec(fdir) - 4;
  found = 0;
  while (!found) {
    x = seek_ec(fdir, eocd);
    assert(x >= 0);
    dir_sig = readintbytesl(fdir, 4);
    if (dir_sig == 101010256) { // 0x06054b50
      found = 1;
    } else {
      eocd = eocd - 1;
    }
  }
  assert(found);
  diskid = readintbytesl(fdir, 2);
  ndisk = readintbytesl(fdir, 2);
  dirid = readintbytesl(fdir, 2);
  ndir = readintbytesl(fdir, 2);

  dir_size = readintbytesl(fdir, 4);
```

```
  dir_start = readintbytesl(fdir, 4);
  eocd_comm_size = readintbytesl(fdir, 2);
  if (eocd_comm_size > 0) {
    eocd_comm = malloc_ec(eocd_comm_size);
    assert(valid(eocd_comm));
    i = 0;
    while (i < eocd_comm_size) {
      b = read_ec(fdir);
      assert(b >= 0);
      eocd_comm[i] = b;
      i = i + 1;
    }
  }

  x = seek_ec(fdir, dir_start);
  assert(x >= 0);
  inspectb (pos_ec(fdir) < eocd && pos_ec(fdir) >=
       0, fdir, 28, 2, 40) {
    dir_sig = readintbytesl(fdir, 4);
    dir_ver_made = readintbytesl(fdir, 2);
    dir_ver_extr = readintbytesl(fdir, 2);
    dir_flag = readintbytesl(fdir, 2);
    dir_comp = readintbytesl(fdir, 2);
    dir_modif = readintbytesl(fdir, 4);
    dir_crc = readintbytesl(fdir, 4);
    dir_ent_size = readintbytesl(fdir, 4);
    dir_ent_size_uncomp = readintbytesl(fdir, 4);
    dir_name_size = readintbytesl(fdir, 2);
    dir_extr_size = readintbytesl(fdir, 2);
    dir_comm_size = readintbytesl(fdir, 2);
    dir_diskid = readintbytesl(fdir, 2);
    dir_attr = malloc_ec(6);
    assert(valid(dir_attr));
    i = 0;
    while (i < 6) {
      b = read_ec(fdir);
      assert(b >= 0);
      dir_attr[i] = b;
      i = i + 1;
    }
    dir_ent_start = readintbytesl(fdir, 4);
    assert(dir_sig == 33639248 && dir_extr_size ==
         24 && dir_comm_size == 0 &&
        dir_name_size > 0);
    dir_name = malloc_ec(dir_name_size);
    assert(valid(dir_name));
    i = 0;
    while (i < dir_name_size) {
      b = read_ec(fdir);
      assert(b >= 0);
      dir_name[i] = b;
      i = i + 1;
    }
    print(dir_name);
    print('\n');
    dir_extr = malloc_ec(dir_extr_size);
    assert(valid(dir_extr));
    i = 0;
    while (i < dir_extr_size) {
      b = read_ec(fdir);
      assert(b >= 0);
      dir_extr[i] = b;
      i = i + 1;
    }

    x = seek_ec(fent, dir_ent_start);
    assert(x >= 0);

    ent_sig = readintbytesl(fent, 4);
    ent_ver_extr = readintbytesl(fent, 2);
    ent_flag = readintbytesl(fent, 2);
    ent_comp = readintbytesl(fent, 2);
    ent_modif = readintbytesl(fent, 4);
    ent_crc = readintbytesl(fent, 4);
    ent_ent_size = readintbytesl(fent, 4);
    ent_ent_size_uncomp = readintbytesl(fent, 4);
    ent_name_size = readintbytesl(fent, 2);
    ent_extr_size = readintbytesl(fent, 2);

    assert(ent_sig == 67324752 && ent_ver_extr ==
         dir_ver_extr && ent_flag == dir_flag &&
         ent_comp == dir_comp && ent_modif ==
         dir_modif && ent_name_size ==
         dir_name_size && ent_name_size > 0);
    ent_name = malloc_ec(ent_name_size);
    assert(valid(ent_name));
    i = 0;
    while (i < ent_name_size) {
      b = read_ec(fent);
      assert(b >= 0);
      ent_name[i] = b;
      i = i + 1;
    }
    print(ent_name);
    print('\n');
    if (ent_extr_size > 0) {
      ent_extr = malloc_ec(ent_extr_size);
      assert(valid(ent_extr));
      i = 0;
      while (i < ent_extr_size) {
        b = read_ec(fent);
        assert(b >= 0);
        ent_extr[i] = b;
        i = i + 1;
      }
    }
    assert(ent_flag & 8 == 0 && ent_crc == dir_crc
         && ent_ent_size == dir_ent_size &&
        ent_ent_size_uncomp ==
        dir_ent_size_uncomp && ent_ent_size > 0);

    raw_data = malloc_ec(ent_ent_size);
    assert(valid(raw_data));
    i = 0;
    while (i < ent_ent_size) {
      b = read_ec(fent);
      assert(b >= 0);
      raw_data[i] = b;
      i = i + 1;
    }
    // decompress
    // check crc
    print(raw_data);
    print('\n');
    x = free_ec(ent_name);
    x = free_ec(raw_data);
  }
  x = seek_ec(fdir, eocd);
  return 0;
}
```

C.5.3 Explicit Recovery Version

```
bad = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x >= 0) {
    if (n == 1) {
      return x;
    } else {
      y = readintbytesl(f, n-1);
      if (bad) {
        return -1;
      }
      return (y << 8) + x;
    }
  } else {
    bad = 1;
    return -1;
  }
}

main {
  fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
  fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
  if (!valid(fdir) || !valid(fent)) {
    exit(1);
  }

  eocd = size_ec(fdir) - 4;
  found = 0;
```

```
  while (!found) {
    x = seek_ec(fdir, eocd);
    if (x < 0) {
      exit(1);
    }
    dir_sig = readintbytesl(fdir, 4);
    if (dir_sig == 101010256) { // 0x06054b50
      found = 1;
    } else {
      eocd = eocd - 1;
    }
  }
  if (!found) {
    exit(1);
  }
  diskid = readintbytesl(fdir, 2);
  ndisk = readintbytesl(fdir, 2);
  dirid = readintbytesl(fdir, 2);
  ndir = readintbytesl(fdir, 2);

  dir_size = readintbytesl(fdir, 4);
  dir_start = readintbytesl(fdir, 4);
  eocd_comm_size = readintbytesl(fdir, 2);
  if (eocd_comm_size > 0) {
    eocd_comm = malloc_ec(eocd_comm_size);
    if (!valid(eocd_comm)) {
      exit(1);
    }
    i = 0;
    while (i < eocd_comm_size) {
      b = read_ec(fdir);
      if (b < 0) {
        exit(1);
      }
      eocd_comm[i] = b;
      i = i + 1;
    }
  }

  x = seek_ec(fdir, dir_start);
  if (x < 0) {
    exit(1);
  }
  lookatb (pos_ec(fdir) < eocd && pos_ec(fdir) >= 0, fdir, 28, 2, 40) {
    bad = 0;
    dir_sig = readintbytesl(fdir, 4);
    dir_ver_made = readintbytesl(fdir, 2);
    dir_ver_extr = readintbytesl(fdir, 2);
    dir_flag = readintbytesl(fdir, 2);
    dir_comp = readintbytesl(fdir, 2);
    dir_modif = readintbytesl(fdir, 4);
    dir_crc = readintbytesl(fdir, 4);
    dir_ent_size = readintbytesl(fdir, 4);
    dir_ent_size_uncomp = readintbytesl(fdir, 4);
    dir_name_size = readintbytesl(fdir, 2);
    dir_extr_size = readintbytesl(fdir, 2);
    dir_comm_size = readintbytesl(fdir, 2);
    dir_diskid = readintbytesl(fdir, 2);
    dir_attr = malloc_ec(6);
    if (valid(dir_attr)) {
      i = 0;
      while (i < 6) {
        b = read_ec(fdir);
        if (b < 0) {
          bad = 1;
        }
        dir_attr[i] = b;
        i = i + 1;
      }
      dir_ent_start = readintbytesl(fdir, 4);
      if (dir_sig == 33639248 && dir_extr_size == 24 && dir_comm_size == 0 && dir_name_size > 0) {
        dir_name = malloc_ec(dir_name_size);
        if (valid(dir_name)) {
          i = 0;
          while (i < dir_name_size) {
            b = read_ec(fdir);
            if (b < 0) {
              bad = 1;
            }
            dir_name[i] = b;
            i = i + 1;
          }
          dir_extr = malloc_ec(dir_extr_size);
          if (valid(dir_extr)) {
            i = 0;
            while (i < dir_extr_size) {
              b = read_ec(fdir);
              if (b < 0) {
                bad = 1;
              }
              dir_extr[i] = b;
              i = i + 1;
            }

            x = seek_ec(fent, dir_ent_start);
            if (x >= 0) {
              ent_sig = readintbytesl(fent, 4);
              ent_ver_extr = readintbytesl(fent, 2);
              ent_flag = readintbytesl(fent, 2);
              ent_comp = readintbytesl(fent, 2);
              ent_modif = readintbytesl(fent, 4);
              ent_crc = readintbytesl(fent, 4);
              ent_ent_size = readintbytesl(fent, 4);
              ent_ent_size_uncomp = readintbytesl(fent, 4);
              ent_name_size = readintbytesl(fent, 2);
              ent_extr_size = readintbytesl(fent, 2);

              print(dir_name);
              print('\n');
              if (ent_sig == 67324752 && ent_ver_extr == dir_ver_extr && ent_flag == dir_flag && ent_comp == dir_comp && ent_modif == dir_modif && ent_name_size == dir_name_size && ent_name_size > 0) {
                ent_name = malloc_ec(ent_name_size);
                if (valid(ent_name)) {
                  i = 0;
                  while (i < ent_name_size) {
                    b = read_ec(fent);
                    if (b < 0) {
                      bad = 1;
                    }
                    ent_name[i] = b;
                    i = i + 1;
                  }
                  if (ent_extr_size > 0) {
                    ent_extr = malloc_ec(ent_extr_size);
                    if (valid(ent_extr)) {
                      i = 0;
                      while (i < ent_extr_size) {
                        b = read_ec(fent);
                        if (b < 0) {
                          bad = 1;
                        }
                        ent_extr[i] = b;
                        i = i + 1;
                      }
                    } else {
                      bad = 1;
                    }
                  }

                  if (ent_flag & 8 == 0 && ent_crc == dir_crc && ent_ent_size == dir_ent_size && ent_ent_size_uncomp == dir_ent_size_uncomp && ent_ent_size > 0) {
                    raw_data = malloc_ec(ent_ent_size);
```

```
                    if (valid(raw_data)) {
                      i = 0;
                      while (i < ent_ent_size) {
                        b = read_ec(fent);
                        if (b < 0) {
                          bad = 1;
                        }
                        raw_data[i] = b;
                        i = i + 1;
                      }
                      // decompress
                      // check crc
                      if (!bad) {
                        print(ent_name);
                        print('\n');
                        print(raw_data);
                        print('\n');
                      } // skip unit
                      x = free_ec(ent_name);
                      x = free_ec(raw_data);
                    } // skip unit
                  } // skip unit
                } // skip unit
              } // skip unit
            } // skip unit
          } // skip unit
        } // skip unit
      } // skip unit
    }
    x = seek_ec(fdir, eocd);
    return 0;
}
```

C.5.4 Conventional Version

```
bad = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x >= 0) {
    if (n == 1) {
      return x;
    } else {
      y = readintbytesl(f, n-1);
      if (bad) {
        return -1;
      }
      return (y << 8) + x;
    }
  } else {
    bad = 1;
    return -1;
  }
}

main {
  fdir = openb_ec("../inputs/zip/stuff-1.zip", 1);
  fent = openb_ec("../inputs/zip/stuff-1.zip", 1);
  if (!valid(fdir) || !valid(fent)) {
    exit(1);
  }

  eocd = size_ec(fdir) - 4;
  found = 0;
  while (!found) {
    x = seek_ec(fdir, eocd);
    if (x < 0) {
      exit(1);
    }
    dir_sig = readintbytesl(fdir, 4);
    if (dir_sig == 101010256) { // 0x06054b50
      found = 1;
    } else {
      eocd = eocd - 1;
    }
  }
  if (!found) {
    exit(1);
  }
  diskid = readintbytesl(fdir, 2);
  ndisk = readintbytesl(fdir, 2);
  dirid = readintbytesl(fdir, 2);
  ndir = readintbytesl(fdir, 2);

  dir_size = readintbytesl(fdir, 4);
  dir_start = readintbytesl(fdir, 4);
  eocd_comm_size = readintbytesl(fdir, 2);
  if (eocd_comm_size > 0) {
    eocd_comm = malloc_ec(eocd_comm_size);
    if (!valid(eocd_comm)) {
      exit(1);
    }
    i = 0;
    while (i < eocd_comm_size) {
      b = read_ec(fdir);
      if (b < 0) {
        exit(1);
      }
      eocd_comm[i] = b;
      i = i + 1;
    }
  }

  x = seek_ec(fdir, dir_start);
  if (x < 0) {
    exit(1);
  }
  finish = 0;
  while (!finish && pos_ec(fdir) < eocd && pos_ec(
      fdir) >= 0) {
    bad = 0;
    dir_sig = readintbytesl(fdir, 4);
    dir_ver_made = readintbytesl(fdir, 2);
    dir_ver_extr = readintbytesl(fdir, 2);
    dir_flag = readintbytesl(fdir, 2);
    dir_comp = readintbytesl(fdir, 2);
    dir_modif = readintbytesl(fdir, 4);
    dir_crc = readintbytesl(fdir, 4);
    dir_ent_size = readintbytesl(fdir, 4);
    dir_ent_size_uncomp = readintbytesl(fdir, 4);
    dir_name_size = readintbytesl(fdir, 2);
    eou = pos_ec(fdir) + dir_name_size + 40;
    if (!bad && dir_name_size >= 0) {
      dir_extr_size = readintbytesl(fdir, 2);
      dir_comm_size = readintbytesl(fdir, 2);
      dir_diskid = readintbytesl(fdir, 2);
      dir_attr = malloc_ec(6);
      if (!bad && valid(dir_attr)) {
        i = 0;
        while (i < 6) {
          b = read_ec(fdir);
          if (b < 0) {
            bad = 1;
          }
          dir_attr[i] = b;
          i = i + 1;
        }
        dir_ent_start = readintbytesl(fdir, 4);
        if (!bad && dir_sig == 33639248 &&
            dir_extr_size == 24 && dir_comm_size
            == 0 && dir_name_size > 0) {
          dir_name = malloc_ec(dir_name_size);
          if (valid(dir_name)) {
            i = 0;
            while (i < dir_name_size) {
              b = read_ec(fdir);
              if (b < 0) {
                bad = 1;
              }
              dir_name[i] = b;
              i = i + 1;
            }
            dir_extr = malloc_ec(dir_extr_size);
            if (!bad && valid(dir_extr)) {
              i = 0;
              while (i < dir_extr_size) {
                b = read_ec(fdir);
                if (b < 0) {
                  bad = 1;
                }
                dir_extr[i] = b;
```

```
                i = i + 1;
              }

              x = seek_ec(fent, dir_ent_start);
              if (x >= 0) {
                ent_sig = readintbytesl(fent, 4);
                ent_ver_extr = readintbytesl(fent, 2);
                ent_flag = readintbytesl(fent, 2);
                ent_comp = readintbytesl(fent, 2);
                ent_modif = readintbytesl(fent, 4);
                ent_crc = readintbytesl(fent, 4);
                ent_ent_size = readintbytesl(fent, 4);
                ent_ent_size_uncomp = readintbytesl(fent, 4);
                ent_name_size = readintbytesl(fent, 2);
                ent_extr_size = readintbytesl(fent, 2);

                print(dir_name);
                print('\n');
                if (ent_sig == 67324752 && ent_ver_extr == dir_ver_extr && ent_flag == dir_flag && ent_comp == dir_comp && ent_modif == dir_modif && ent_name_size == dir_name_size && ent_name_size > 0) {
                  ent_name = malloc_ec(ent_name_size);
                  if (valid(ent_name)) {
                    i = 0;
                    while (i < ent_name_size) {
                      b = read_ec(fent);
                      if (b < 0) {
                        bad = 1;
                      }
                      ent_name[i] = b;
                      i = i + 1;
                    }
                    if (ent_extr_size > 0) {
                      ent_extr = malloc_ec(ent_extr_size);
                      if (valid(ent_extr)) {
                        i = 0;
                        while (i < ent_extr_size) {
                          b = read_ec(fent);
                          if (b < 0) {
                            bad = 1;
                          }
                          ent_extr[i] = b;
                          i = i + 1;
                        }
                      } else {
                        bad = 1;
                      }
                    }

                    if (ent_flag & 8 == 0 && ent_crc == dir_crc && ent_ent_size == dir_ent_size && ent_ent_size_uncomp == dir_ent_size_uncomp && ent_ent_size > 0) {
                      raw_data = malloc_ec(ent_ent_size);
                      if (valid(raw_data)) {
                        i = 0;
                        while (i < ent_ent_size) {
                          b = read_ec(fent);
                          if (b < 0) {
                            bad = 1;
                          }
                          raw_data[i] = b;
                          i = i + 1;
                        }
                        // decompress
                        // check crc
                        if (!bad) {
                          print(ent_name);
                          print('\n');
                          print(raw_data);
                          print('\n');
                        } // skip unit
                        x = free_ec(ent_name);
                        x = free_ec(raw_data);
                      } // skip unit
                    } // skip unit
                  } // skip unit
                } // skip unit
              } // skip unit
            } // skip unit
          } // skip unit
        } // skip unit
      } // skip unit
      x = seek_ec(fdir, eou);
      if (x < 0) { // give up
        finish = 1;
      }
    } else { // give up
      finish = 1;
    }
  }
  x = seek_ec(fdir, eocd);
  return 0;
}
```

C.6 RGIF

RGIF files store animated images. RGIF is based on the GIF format. An GIF file contains header information followed by blocks. Each block starts with one or two bytes that describe the block type, such as image data, metadata, or other extensions. Each image may contain several consecutive blocks. The RGIF format differs from GIF only in that it adds two bytes in front of each image to describe the total length of all the blocks for the image.

The benchmark program reads an RGIF file and converts it to the GIF format. When an image is corrupted, the program discards the image and keeps parsing. The program rejects RGIF files with malformed headers. The program discards trailing inputs after seeing an image with negative length.

C.6.1 Full RIFL Version

```
buffer = malloc(7500);
number = 0;

func readintbytesl (f, n) { // little endian
  x = read(f);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readblocks (f, echo) {
  i = 0;
  stop = 0;
  lookatb (!stop, f, 0, 1, 0) {
    len = read(f);
    if (echo) {
      print(len);
    }
    if (len == 0) {
      stop = 1;
```

```
    } else {
      while (!end(f)) {
        x = read(f);
        if (echo) {
          print(x);
        }
        buffer[i] = x;
        i = i + 1;
      }
    }
  }
  assert(stop);
  return i;
}

func printintbytesl (n) {
  b = number & 255;
  print(b);
  if (n == 1) {
    return 0;
  } else {
    number = number >> 8;
    dummy = printintbytesl(n - 1);
    return 0;
  }
}

main {
  f = openb("../inputs/rgif/welcome2-block.rgif",
      1);

  // header block
  header = malloc(6);
  i = 0;
  while (i < 6) {
    x = read(f);
    print(x);
    header[i] = x;
    i = i + 1;
  }
  assert(header[0] == 'G' && header[1] == 'I' &&
      header[2] == 'F' && header[3] == '8' && (
      header[4] == '9' || header[4] == '7') &&
      header[5] == 'a');
  free(header);

  // logical screen descriptor
  canvasw = readintbytesl(f, 2);
  number = canvasw;
  dummy = printintbytesl(2);
  canvash = readintbytesl(f, 2);
  number = canvash;
  dummy = printintbytesl(2);
  x = read(f);
  print(x);
  gflag = (x & 128) >> 7;
  bpp = (x & 112) >> 4;
  gsort = (x & 8) >> 3;
  background = read(f);
  print(background);
  aspect = read(f);
  print(aspect);
  assert(gflag && x & 7 == bpp);

  // global color table
  nglobal = 2 << bpp;
  gcolors = malloc(nglobal);
  i = 0;
  while (i < nglobal) {
    x = readintbytesl(f, 3);
    number = x;
    dummy = printintbytesl(3);
    gcolors[i] = x;
    i = i + 1;
  }

  trailer = 0;
  inspectb (!trailer, f, 0, 2, 1) {
    length = readintbytesl(f, 2);
    x = read(f);
    if (x == 59) { // trailer block
      print(x);
      trailer = 1;
    } else {
      label = read(f);
      if (x == 33 && (label == 255 || label ==
          254)) {
        if (label == 255) { // appliation extension
          idlen = read(f);
          id = malloc(idlen);
          i = 0;
          while (i < idlen) {
            x = read(f);
            id[i] = x;
            i = i + 1;
          }
          free(id);
          applen = readblocks(f, 0);
          app = malloc(applen);
          i = 0;
          while (i < applen) {
            app[i] = buffer[i];
            i = i + 1;
          }
        } else { // comment extension
          commentlen = readblocks(f, 0);
          comments = malloc(commentlen);
          i = 0;
          while (i < commentlen) {
            comments[i] = buffer[i];
            i = i + 1;
          }
          free(comments);
        }
      } else {
        if (x == 33 && label == 249) { // graphics
            control extension
          print(x);
          print(label);
          blocksize = read(f);
          print(blocksize);
          ctrlext = read(f);
          print(ctrlext);
          delay = readintbytesl(f, 2);
          number = delay;
          dummy = printintbytesl(2);
          transparent = read(f);
          print(transparent);
          terminator = read(f);
          print(terminator);
          assert(terminator == 0);
        } else {
          assert(x != 33);
          seek(f, pos(f)-2);
        }

        // image descriptor
        x = read(f);
        assert(x == 44);
        print(x);
        left = readintbytesl(f, 2);
        number = left;
        dummy = printintbytesl(2);
        top = readintbytesl(f, 2);
        number = top;
        dummy = printintbytesl(2);
        width = readintbytesl(f, 2);
        number = width;
        dummy = printintbytesl(2);
        height = readintbytesl(f, 2);
        number = height;
        dummy = printintbytesl(2);
        assert(left == 0 && top == 0 && width ==
            canvasw && height == canvash);
        x = read(f);
        print(x);
        lflag = (x & 128) >> 7;
        linterlace = (x & 64) >> 6;
        lsort = (x & 32) >> 5;
        lctsize = x & 7;

        // local color table
```

```
        if (lflag) {
          nlocal = 2 << bpp;
          lcolors = malloc(nlocal);
          i = 0;
          while (i < nlocal) {
            x = readintbytesl(f, 3);
            number = x;
            dummy = printintbytesl(3);
            lcolors[i] = x;
            i = i + 1;
          }
        }

        // data sub-blocks
        lzw = read(f);
        print(lzw);
        datalen = readblocks(f, 1);
        // decompress
        // decode
      }
    }
  }
  free(buffer);
  if (!trailer) {
    x = 59;
    print(x);
  }
  return 0;
}
```

C.6.2 Explicit Check Version

```
buffer = malloc_ec(7500);
number = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  assert(x >= 0);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readblocks (f, echo) {
  i = 0;
  stop = 0;
  lookatb (!stop, f, 0, 1, 0) {
    len = read_ec(f);
    assert(len >= 0);
    if (echo) {
      print(len);
    }
    if (len == 0) {
      stop = 1;
    } else {
      while (!end_ec(f)) {
        x = read_ec(f);
        assert(x >= 0 && i < 7500);
        if (echo) {
          print(x);
        }
        buffer[i] = x;
        i = i + 1;
      }
    }
  }
  assert(stop);
  return i;
}

func printintbytesl (n) {
  b = number & 255;
  print(b);
  if (n == 1) {
    return 0;
  } else {
    number = number >> 8;
    dummy = printintbytesl(n - 1);
    return 0;
  }
}

main {
  f = openb_ec("../inputs/rgif/welcome2-block.rgif
      ", 1);
  assert(valid(buffer) && valid(f));

  // header block
  header = malloc_ec(6);
  assert(valid(header));
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    assert(x >= 0);
    print(x);
    header[i] = x;
    i = i + 1;
  }
  assert(header[0] == 'G' && header[1] == 'I' &&
      header[2] == 'F' && header[3] == '8' && (
      header[4] == '9' || header[4] == '7') &&
      header[5] == 'a');
  x = free_ec(header);

  // logical screen descriptor
  canvasw = readintbytesl(f, 2);
  number = canvasw;
  dummy = printintbytesl(2);
  canvash = readintbytesl(f, 2);
  number = canvash;
  dummy = printintbytesl(2);
  x = read_ec(f);
  assert(x >= 0);
  print(x);
  gflag = (x & 128) >> 7;
  bpp = (x & 112) >> 4;
  gsort = (x & 8) >> 3;
  background = read_ec(f);
  print(background);
  aspect = read_ec(f);
  print(aspect);
  assert(gflag && x & 7 == bpp && background >= 0
      && aspect >= 0);

  // global color table
  nglobal = 2 << bpp;
  gcolors = malloc_ec(nglobal);
  assert(valid(gcolors));
  i = 0;
  while (i < nglobal) {
    x = readintbytesl(f, 3);
    number = x;
    dummy = printintbytesl(3);
    gcolors[i] = x;
    i = i + 1;
  }

  trailer = 0;
  inspectb (!trailer, f, 0, 2, 1) {
    length = readintbytesl(f, 2);
    x = read_ec(f);
    assert(x >= 0);
    if (x == 59) { // trailer block
      print(x);
      trailer = 1;
    } else {
      label = read_ec(f);
      assert(label >= 0);
      if (x == 33 && (label == 255 || label ==
          254)) {
        if (label == 255) { // appliation extension
          idlen = read_ec(f);
          assert(idlen >= 0);
          id = malloc_ec(idlen);
          assert(valid(id));
          i = 0;
          while (i < idlen) {
            x = read_ec(f);
            assert(x >= 0);
```

```
            id[i] = x;
            i = i + 1;
          }
          x = free_ec(id);
          applen = readblocks(f, 0);
          app = malloc_ec(applen);
          assert(valid(app));
          i = 0;
          while (i < applen) {
            app[i] = buffer[i];
            i = i + 1;
          }
        } else { // comment extension
          commentlen = readblocks(f, 0);
          comments = malloc_ec(commentlen);
          assert(valid(comments));
          i = 0;
          while (i < commentlen) {
            comments[i] = buffer[i];
            i = i + 1;
          }
          x = free_ec(comments);
        }
      } else {
        if (x == 33 && label == 249) { // graphics
            control extension
          print(x);
          print(label);
          blocksize = read_ec(f);
          print(blocksize);
          ctrlext = read_ec(f);
          print(ctrlext);
          delay = readintbytesl(f, 2);
          number = delay;
          dummy = printintbytesl(2);
          transparent = read_ec(f);
          print(transparent);
          terminator = read_ec(f);
          print(terminator);
          assert(blocksize >= 0 && ctrlext >= 0 &&
                transparent >= 0 && terminator ==
               0);
        } else {
          assert(x != 33);
          x = seek_ec(f, pos_ec(f)-2);
          assert(x >= 0);
        }

        // image descriptor
        x = read_ec(f);
        assert(x == 44);
        print(x);
        left = readintbytesl(f, 2);
        number = left;
        dummy = printintbytesl(2);
        top = readintbytesl(f, 2);
        number = top;
        dummy = printintbytesl(2);
        width = readintbytesl(f, 2);
        number = width;
        dummy = printintbytesl(2);
        height = readintbytesl(f, 2);
        number = height;
        dummy = printintbytesl(2);
        assert(left == 0 && top == 0 && width ==
             canvasw && height == canvash);
        x = read_ec(f);
        assert(x >= 0);
        print(x);
        lflag = (x & 128) >> 7;
        linterlace = (x & 64) >> 6;
        lsort = (x & 32) >> 5;
        lctsize = x & 7;

        // local color table
        if (lflag) {
          nlocal = 2 << bpp;
          lcolors = malloc_ec(nlocal);
          assert(valid(lcolors));
          i = 0;
          while (i < nlocal) {
            x = readintbytesl(f, 3);
            number = x;
            dummy = printintbytesl(3);
            lcolors[i] = x;
            i = i + 1;
          }
        }

        // data sub-blocks
        lzw = read_ec(f);
        assert(lzw >= 0);
        print(lzw);
        datalen = readblocks(f, 1);
        // decompress
        // decode
      }
    }
  }
  x = free_ec(buffer);
  if (!trailer) {
    x = 59;
    print(x);
  }
  return 0;
}
```

C.6.3 Explicit Recovery Version

```
buffer = malloc_ec(7500);
number = 0;
prt = 1;

bad = 0;
out = malloc_ec(7500);
outidx = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    if (bad) {
      return -1;
    }
    return (y << 8) + x;
  }
}

func readblocks (f, echo) {
  i = 0;
  stop = 0;
  lookatb (!stop, f, 0, 1, 0) {
    len = read_ec(f);
    if (len >= 0) {
      if (echo) {
        dummy = write(len);
      }
      if (len == 0) {
        stop = 1;
      } else {
        while (!end_ec(f) && !bad) {
          x = read_ec(f);
          if (x >= 0 && i < 7500) {
            if (echo) {
              dummy = write(x);
            }
            buffer[i] = x;
            i = i + 1;
          } else {
            bad = 1;
          }
        }
      }
    } else {
      bad = 1;
    }
```

```
  }
  if (!stop) {
    bad = 1;
  }
  return i;
}

func write (x) {
  if (outidx < 7500) {
    out[outidx] = x;
    outidx = outidx + 1;
    return 0;
  } else {
    bad = 1;
    return -1;
  }
}

func writeintbytesl (n) {
  b = number & 255;
  if (prt) {
    print(b);
  } else {
    dummy = write(b);
  }
  if (n == 1) {
    return 0;
  } else {
    number = number >> 8;
    dummy = writeintbytesl(n - 1);
    return 0;
  }
}

main {
  f = openb_ec("../inputs/rgif/welcome2-block.rgif
      ", 1);
  if (!valid(buffer) || !valid(out) || !valid(f))
      {
    exit(1);
  }

  // header block
  header = malloc_ec(6);
  if (!valid(header)) {
    exit(1);
  }
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    if (x < 0) {
      exit(1);
    }
    print(x);
    header[i] = x;
    i = i + 1;
  }
  if (!(header[0] == 'G' && header[1] == 'I' &&
      header[2] == 'F' && header[3] == '8' && (
      header[4] == '9' || header[4] == '7') &&
      header[5] == 'a')) {
    exit(1);
  }
  x = free_ec(header);

  // logical screen descriptor
  canvasw = readintbytesl(f, 2);
  number = canvasw;
  dummy = writeintbytesl(2);
  canvash = readintbytesl(f, 2);
  number = canvash;
  dummy = writeintbytesl(2);
  x = read_ec(f);
  if (bad || x < 0) {
    exit(1);
  }
  print(x);
  gflag = (x & 128) >> 7;
  bpp = (x & 112) >> 4;
  gsort = (x & 8) >> 3;
  background = read_ec(f);
  print(background);
  aspect = read_ec(f);
  print(aspect);
  if (!gflag || x & 7 != bpp || background < 0 ||
      aspect < 0) {
    exit(1);
  }

  // global color table
  nglobal = 2 << bpp;
  gcolors = malloc_ec(nglobal);
  if (!valid(gcolors)) {
    exit(1);
  }
  i = 0;
  while (i < nglobal) {
    x = readintbytesl(f, 3);
    if (bad) {
      exit(1);
    }
    number = x;
    dummy = writeintbytesl(3);
    gcolors[i] = x;
    i = i + 1;
  }

  prt = 0;
  trailer = 0;
  lookatb (!trailer, f, 0, 2, 1) {
    bad = 0;
    outidx = 0;
    length = readintbytesl(f, 2);
    x = read_ec(f);
    if (x >= 0) {
      if (x == 59) { // trailer block
        dummy = write(x);
        trailer = 1;
      } else {
        label = read_ec(f);
        if (label < 0) {
          bad = 1;
        }
        if (x == 33 && (label == 255 || label ==
            254)) {
          if (label == 255) { // appliation extension
            idlen = read_ec(f);
            if (idlen < 0) {
              bad = 1;
            }
            id = malloc_ec(idlen);
            if (valid(id)) {
              i = 0;
              while (i < idlen) {
                x = read_ec(f);
                if (x < 0) {
                  bad = 1;
                }
                id[i] = x;
                i = i + 1;
              }
              x = free_ec(id);
              applen = readblocks(f, 0);
              app = malloc_ec(applen);
              if (valid(app)) {
                i = 0;
                while (i < applen) {
                  app[i] = buffer[i];
                  i = i + 1;
                }
              } else {
                bad = 1;
              }
            } else {
              bad = 1;
            }
          } else { // comment extension
            commentlen = readblocks(f, 0);
            comments = malloc_ec(commentlen);
            if (valid(comments)) {
              i = 0;
              while (i < commentlen) {
```

```
                comments[i] = buffer[i];
                i = i + 1;
              }
              x = free_ec(comments);
            } else {
              bad = 1;
            }
          }
        } else {
          if (x == 33 && label == 249) { //
               graphics control extension
            dummy = write(x);
            dummy = write(label);
            blocksize = read_ec(f);
            dummy = write(blocksize);
            ctrlext = read_ec(f);
            dummy = write(ctrlext);
            delay = readintbytesl(f, 2);
            number = delay;
            dummy = writeintbytesl(2);
            transparent = read_ec(f);
            dummy = write(transparent);
            terminator = read_ec(f);
            dummy = write(terminator);
            if (blocksize < 0 || ctrlext < 0 ||
                 transparent < 0 || terminator !=
                 0) {
              bad = 1;
            }
          } else {
            if (x == 33) {
              bad = 1;
            }
            x = seek_ec(f, pos_ec(f)-2);
            if (x < 0) {
              bad = 1;
            }
          }

          // image descriptor
          x = read_ec(f);
          if (x != 44) {
            bad = 1;
          }
          dummy = write(x);
          left = readintbytesl(f, 2);
          number = left;
          dummy = writeintbytesl(2);
          top = readintbytesl(f, 2);
          number = top;
          dummy = writeintbytesl(2);
          width = readintbytesl(f, 2);
          number = width;
          dummy = writeintbytesl(2);
          height = readintbytesl(f, 2);
          number = height;
          dummy = writeintbytesl(2);
          if (left != 0 || top != 0 || width !=
               canvasw || height != canvash) {
            bad = 1;
          }
          x = read_ec(f);
          if (x < 0) {
            bad = 1;
          }
          dummy = write(x);
          lflag = (x & 128) >> 7;
          linterlace = (x & 64) >> 6;
          lsort = (x & 32) >> 5;
          lctsize = x & 7;

          // local color table
          if (lflag) {
            nlocal = 2 << bpp;
            lcolors = malloc_ec(nlocal);
            if (valid(lcolors)) {
              i = 0;
              while (i < nlocal) {
                x = readintbytesl(f, 3);
                number = x;
                dummy = writeintbytesl(3);
                lcolors[i] = x;
                i = i + 1;
              }
            } else {
              bad = 1;
            }
          }

          // data sub-blocks
          lzw = read_ec(f);
          if (lzw < 0) {
            bad = 1;
          }
          dummy = write(lzw);
          datalen = readblocks(f, 1);
          // decompress
          // decode
        }
      }
    } else {
      bad = 1;
    }
    if (!bad) {
      i = 0;
      while (i < outidx) {
        x = out[i];
        print(x);
        i = i + 1;
      }
    } // skip unit
  }
  x = free_ec(buffer);
  if (!trailer) {
    x = 59;
    print(x);
  }
  return 0;
}
```

C.6.4 Conventional Version

```
buffer = malloc_ec(7500);
number = 0;
prt = 1;

bad = 0;
out = malloc_ec(7500);
outidx = 0;

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    if (bad) {
      return -1;
    }
    return (y << 8) + x;
  }
}

func readblocks (f, echo) {
  i = 0;
  stop = 0;
  while (!stop) {
    len = read_ec(f);
    if (len >= 0) {
      if (echo) {
        dummy = write(len);
      }
      if (len == 0) {
        stop = 1;
      } else {
        j = 0;
        while (j < len) {
          x = read_ec(f);
```

```
            if (x >= 0 && i < 7500) {
              if (echo) {
                dummy = write(x);
              }
              buffer[i] = x;
              i = i + 1;
            } else {
              bad = 1;
            }
            j = j + 1;
          }
        }
      } else {
        bad = 1;
      }
    }
    return i;
  }

  func write (x) {
    if (outidx < 7500) {
      out[outidx] = x;
      outidx = outidx + 1;
      return 0;
    } else {
      bad = 1;
      return -1;
    }
  }

  func writeintbytesl (n) {
    b = number & 255;
    if (prt) {
      print(b);
    } else {
      dummy = write(b);
    }
    if (n == 1) {
      return 0;
    } else {
      number = number >> 8;
      dummy = writeintbytesl(n - 1);
      return 0;
    }
  }

  main {
    f = openb_ec("../inputs/rgif/welcome2-block.rgif
        ", 1);
    if (!valid(buffer) || !valid(out) || !valid(f))
        {
      exit(1);
    }

    // header block
    header = malloc_ec(6);
    if (!valid(header)) {
      exit(1);
    }
    i = 0;
    while (i < 6) {
      x = read_ec(f);
      if (x < 0) {
        exit(1);
      }
      print(x);
      header[i] = x;
      i = i + 1;
    }
    if (!(header[0] == 'G' && header[1] == 'I' &&
         header[2] == 'F' && header[3] == '8' && (
         header[4] == '9' || header[4] == '7') &&
         header[5] == 'a')) {
      exit(1);
    }
    x = free_ec(header);

    // logical screen descriptor
    canvasw = readintbytesl(f, 2);
    number = canvasw;
    dummy = writeintbytesl(2);
    canvash = readintbytesl(f, 2);
    number = canvash;
    dummy = writeintbytesl(2);
    x = read_ec(f);
    if (bad || x < 0) {
      exit(1);
    }
    print(x);
    gflag = (x & 128) >> 7;
    bpp = (x & 112) >> 4;
    gsort = (x & 8) >> 3;
    background = read_ec(f);
    print(background);
    aspect = read_ec(f);
    print(aspect);
    if (!gflag || x & 7 != bpp || background < 0 ||
        aspect < 0) {
      exit(1);
    }

    // global color table
    nglobal = 2 << bpp;
    gcolors = malloc_ec(nglobal);
    if (!valid(gcolors)) {
      exit(1);
    }
    i = 0;
    while (i < nglobal) {
      x = readintbytesl(f, 3);
      if (bad) {
        exit(1);
      }
      number = x;
      dummy = writeintbytesl(3);
      gcolors[i] = x;
      i = i + 1;
    }

    prt = 0;
    trailer = 0;
    finish = 0;
    while (!trailer && !finish) {
      bad = 0;
      outidx = 0;
      length = readintbytesl(f, 2);
      if (length >= 0) {
        eou = pos_ec(f) + length + 1;
        x = read_ec(f);
        if (x >= 0) {
          if (x == 59) { // trailer block
            dummy = write(x);
            trailer = 1;
          } else {
            label = read_ec(f);
            if (label < 0) {
              bad = 1;
            }
            if (x == 33 && (label == 255 || label ==
                254)) {
              if (label == 255) { // appliation
                  extension
                idlen = read_ec(f);
                if (idlen < 0) {
                  bad = 1;
                }
                id = malloc_ec(idlen);
                if (valid(id)) {
                  i = 0;
                  while (i < idlen) {
                    x = read_ec(f);
                    if (x < 0) {
                      bad = 1;
                    }
                    id[i] = x;
                    i = i + 1;
                  }
                  x = free_ec(id);
                  applen = readblocks(f, 0);
                  app = malloc_ec(applen);
                  if (valid(app)) {
                    i = 0;
```

```
                while (i < applen) {
                  app[i] = buffer[i];
                  i = i + 1;
                }
              } else {
                bad = 1;
              }
            } else {
              bad = 1;
            }
          } else { // comment extension
            commentlen = readblocks(f, 0);
            comments = malloc_ec(commentlen);
            if (valid(comments)) {
              i = 0;
              while (i < commentlen) {
                comments[i] = buffer[i];
                i = i + 1;
              }
              x = free_ec(comments);
            } else {
              bad = 1;
            }
          }
        } else {
          if (x == 33 && label == 249) { //
               graphics control extension
            dummy = write(x);
            dummy = write(label);
            blocksize = read_ec(f);
            dummy = write(blocksize);
            ctrlext = read_ec(f);
            dummy = write(ctrlext);
            delay = readintbytesl(f, 2);
            number = delay;
            dummy = writeintbytesl(2);
            transparent = read_ec(f);
            dummy = write(transparent);
            terminator = read_ec(f);
            dummy = write(terminator);
            if (blocksize < 0 || ctrlext < 0 ||
                 transparent < 0 || terminator
                 != 0) {
              bad = 1;
            }
          } else {
            if (x == 33) {
              bad = 1;
            }
            x = seek_ec(f, pos_ec(f)-2);
            if (x < 0) {
              bad = 1;
            }
          }

          // image descriptor
          x = read_ec(f);
          if (x != 44) {
            bad = 1;
          }
          dummy = write(x);
          left = readintbytesl(f, 2);
          number = left;
          dummy = writeintbytesl(2);
          top = readintbytesl(f, 2);
          number = top;
          dummy = writeintbytesl(2);
          width = readintbytesl(f, 2);
          number = width;
          dummy = writeintbytesl(2);
          height = readintbytesl(f, 2);
          number = height;
          dummy = writeintbytesl(2);
          if (left != 0 || top != 0 || width !=
               canvasw || height != canvash) {
            bad = 1;
          }
          x = read_ec(f);
          if (x < 0) {
            bad = 1;
          }
          dummy = write(x);
          lflag = (x & 128) >> 7;
          linterlace = (x & 64) >> 6;
          lsort = (x & 32) >> 5;
          lctsize = x & 7;

          // local color table
          if (lflag) {
            nlocal = 2 << bpp;
            lcolors = malloc_ec(nlocal);
            if (valid(lcolors)) {
              i = 0;
              while (i < nlocal) {
                x = readintbytesl(f, 3);
                number = x;
                dummy = writeintbytesl(3);
                lcolors[i] = x;
                i = i + 1;
              }
            } else {
              bad = 1;
            }
          }

          // data sub-blocks
          lzw = read_ec(f);
          if (lzw < 0) {
            bad = 1;
          }
          dummy = write(lzw);
          datalen = readblocks(f, 1);
          // decompress
          // decode
        }
      }
    } else {
      bad = 1;
    }
    if (pos_ec(f) > eou || pos_ec(f) < 0) {
      bad = 1;
    }
    x = seek_ec(f, eou);
    if (x < 0) { // give up
      finish = 1;
    }
    if (!bad && !finish) {
      i = 0;
      while (i < outidx) {
        x = out[i];
        print(x);
        i = i + 1;
      }
    } // skip unit
  } else { // give up
    finish = 1;
  }
}
x = free_ec(buffer);
if (!trailer) {
  x = 59;
  print(x);
}
return 0;
}
```

C.7 PCAP/DNS

PCAP files store network packets. A PCAP file contains a header and capture items. Each capture item contains 12 bytes of metadata, a 4-byte length field, and a network packet of "length" bytes. The PCAP/DNS benchmark considers only DNS packets. Specifically, each network packet of interest contains an Ethernet header, an Internet Protocol version 4 (IPv4) header, a User Datagram Protocol (UDP) header, the DNS packet, and some trailing Ethernet bytes.

The benchmark program reads a PCAP file and extracts DNS packets. For each valid DNS packet, the program prints

the source Internet Protocol (IP) address, the destination IP address, the DNS identification number, and the DNS questions and answers. When a network packet is malformed, the program discards the capture item and keeps parsing. The program rejects PCAP files with malformed headers. The program discards trailing inputs after seeing a capture item with negative length.

C.7.1 Full RIFL Version

```
dst_mac = malloc(6);
src_mac = malloc(6);
data = malloc(1);

func readintbytesl (f, n) { // little endian
  x = read(f);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readintbytesb (f, n) { // big endian
  x = 0;
  i = 0;
  while (i < n) {
    byte = read(f);
    x = x << 8;
    x = x | byte;
    i = i + 1;
  }
  return x;
}

func parse_ethernet (f, dummy) {
  i = 0;
  while (i < 6) {
    x = read(f);
    dst_mac[i] = x;
    i = i + 1;
  }
  i = 0;
  while (i < 6) {
    x = read(f);
    src_mac[i] = x;
    i = i + 1;
  }

  ether_type = readintbytesl(f, 2);
  assert(ether_type == 8); // IPv4

  dummy = parse_ipv4(f, 0);
  return 0;
}

func check_sum (f, len) {
  start = pos(f);
  assert(len > 0 && len % 2 == 0);
  sum = 0;
  i = 0;
  while (i < len / 2) {
    x = readintbytesb(f, 2);
    sum = sum + x;
    i = i + 1;
  }
  hi = (sum >> 16) & 65535;
  lo = sum & 65535;
  assert(hi + lo == 65535);
  seek(f, start);
  return 0;
}

func print_ip (addr) {
  b = (addr >> 24) & 255;
  print(b);
  print('.');
  b = (addr >> 16) & 255;
  print(b);
  print('.');
  b = (addr >> 8) & 255;
  print(b);
  print('.');
  b = addr & 255;
  print(b);
  return 0;
}

func parse_ipv4 (f, dummy) {
  dummy = check_sum(f, 20); // no options

  x = read(f);
  version = (x & 240) >> 4;
  hdr_size = x & 15;
  x = read(f);
  total = readintbytesb(f, 2);

  id = readintbytesb(f, 2);
  fragment = readintbytesb(f, 2);
  dont_frag = (fragment & 16384) >> 14;
  more_frag = (fragment & 8192) >> 13;
  offs_frag = fragment & 8191;

  ttl = read(f);
  ip_type = read(f);
  checksum = readintbytesb(f, 2);

  src_ip = readintbytesb(f, 4);
  dst_ip = readintbytesb(f, 4);

  assert(version == 4 && hdr_size == 5 && total >=
       hdr_size * 4 && fragment & 32768 == 0 &&
      more_frag == 0 && offs_frag == 0 && ip_type
       == 17); // IPv4, no options,
      UDP

  print('\n');
  dummy = print_ip(src_ip);
  print(' ');
  dummy = print_ip(dst_ip);
  print(' ');
  dummy = parse_udp(f, total - hdr_size * 4);

  return 0;
}

func parse_udp (f, pkt_size) {
  src_port = readintbytesb(f, 2);
  dst_port = readintbytesb(f, 2);
  udp_size = readintbytesb(f, 2);
  checksum = readintbytesb(f, 2);

  assert(pkt_size <= udp_size && pkt_size > 8);
  if (src_port == 53 || dst_port == 53) { // DNS
    dummy = parse_dns(f, pkt_size - 8);
  } else { // others
    if (valid(data)) {
      free(data);
    }
    data_size = pkt_size - 8;
    data = malloc(data_size);
    i = 0;
    while (i < data_size) {
      x = read(f);
      data[i] = x;
      i = i + 1;
    }
    print('\n');
  }
  return 0;
}

func parse_dns (f, data_size) {
  id = readintbytesb(f, 2);
  print(id);
  print('\n');
  flags = readintbytesb(f, 2);
```

```
  n_q = readintbytesb(f, 2);
  n_ans = readintbytesb(f, 2);
  n_auth = readintbytesb(f, 2);
  n_add = readintbytesb(f, 2);

  // questions
  i = 0;
  while (i < n_q) {
    stop = 0;
    first = 1;
    while (!stop) {
      len = read(f);
      if (len == 0) {
        stop = 1;
      } else {
        if (first) {
          first = 0;
        } else {
          print('.');
        }
        j = 0;
        while (j < len) {
          x = read(f);
          print(x);
          j = j + 1;
        }
      }
    }
    print('\n');
    type_q = readintbytesb(f, 2);
    class_q = readintbytesb(f, 2);
    i = i + 1;
  }

  // answers
  i = 0;
  while (i < n_ans) {
    ptr = readintbytesb(f, 2);
    assert(ptr == 49164);
    type_ans = readintbytesb(f, 2);
    class_ans = readintbytesb(f, 2);
    ttl = readintbytesb(f, 4);
    len = readintbytesb(f, 2);
    if (type_ans == 1) {
      assert(len == 4);
      addr = readintbytesb(f, 4);
      dummy = print_ip(addr);
    } else {
      assert(type_ans == 5);
      j = 0;
      while (j < len) {
        x = read(f);
        print(x);
        j = j + 1;
      }
    }
    print('\n');
    i = i + 1;
  }

  // authority
  // additional
  return 0;
}

main {
  f = openb("../inputs/pcap/bad.pcap", 1);

  x = read(f);
  assert(x == 212);
  x = read(f);
  assert(x == 195);
  x = read(f);
  assert(x == 178);
  x = read(f);
  assert(x == 161);
  maj_ver = readintbytesl(f, 2);
  min_ver = readintbytesl(f, 2);
  this_zone = readintbytesl(f, 4);
  sigfigs = readintbytesl(f, 4);
  snap_len = readintbytesl(f, 4); // number of
      packets
  link_type = readintbytesl(f, 4);
  assert(link_type == 1); // ethernet

  inspectb (1, f, 12, 4, 0) {
    ts_epoch = readintbytesl(f, 4);
    ts_nanosec = readintbytesl(f, 4);
    caplen = readintbytesl(f, 4);
    length = readintbytesl(f, 4);
    assert(caplen == length);

    dummy = parse_ethernet(f, 0);
  }

  return 0;
}
```

C.7.2 Explicit Check Version

```
dst_mac = malloc_ec(6);
src_mac = malloc_ec(6);
data = malloc_ec(1);

func readintbytesl (f, n) { // little endian
  x = read_ec(f);
  assert(x >= 0);
  if (n == 1) {
    return x;
  } else {
    y = readintbytesl(f, n-1);
    return (y << 8) + x;
  }
}

func readintbytesb (f, n) { // big endian
  x = 0;
  i = 0;
  while (i < n) {
    byte = read_ec(f);
    assert(byte >= 0);
    x = x << 8;
    x = x | byte;
    i = i + 1;
  }
  return x;
}

func parse_ethernet (f, dummy) {
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    assert(x >= 0);
    dst_mac[i] = x;
    i = i + 1;
  }
  i = 0;
  while (i < 6) {
    x = read_ec(f);
    assert(x >= 0);
    src_mac[i] = x;
    i = i + 1;
  }

  ether_type = readintbytesl(f, 2);
  assert(ether_type == 8); // IPv4

  dummy = parse_ipv4(f, 0);
  return 0;
}

func check_sum (f, len) {
  start = pos_ec(f);
  assert(len > 0 && len % 2 == 0 && start >= 0);
  sum = 0;
  i = 0;
  while (i < len / 2) {
    x = readintbytesb(f, 2);
    sum = sum + x;
    i = i + 1;
  }
```

```
  hi = (sum >> 16) & 65535;
  lo = sum & 65535;
  x = seek_ec(f, start);
  assert(hi + lo == 65535 && x >= 0);
  return 0;
}

func print_ip (addr) {
  b = (addr >> 24) & 255;
  print(b);
  print('.');
  b = (addr >> 16) & 255;
  print(b);
  print('.');
  b = (addr >> 8) & 255;
  print(b);
  print('.');
  b = addr & 255;
  print(b);
  return 0;
}

func parse_ipv4 (f, dummy) {
  dummy = check_sum(f, 20); // no options

  x = read_ec(f);
  assert(x >= 0);
  version = (x & 240) >> 4;
  hdr_size = x & 15;
  x = read_ec(f);
  assert(x >= 0);
  total = readintbytesb(f, 2);

  id = readintbytesb(f, 2);
  fragment = readintbytesb(f, 2);
  dont_frag = (fragment & 16384) >> 14;
  more_frag = (fragment & 8192) >> 13;
  offs_frag = fragment & 8191;

  ttl = read_ec(f);
  ip_type = read_ec(f);
  checksum = readintbytesb(f, 2);

  src_ip = readintbytesb(f, 4);
  dst_ip = readintbytesb(f, 4);

  assert(version == 4 && hdr_size == 5 && total >=
       hdr_size * 4 && fragment & 32768 == 0 &&
      more_frag == 0 && offs_frag == 0 && ttl >=
      0 && ip_type == 17); // IPv4, no options,
      UDP

  print('\n');
  dummy = print_ip(src_ip);
  print(' ');
  dummy = print_ip(dst_ip);
  print(' ');
  dummy = parse_udp(f, total - hdr_size * 4);

  return 0;
}

func parse_udp (f, pkt_size) {
  src_port = readintbytesb(f, 2);
  dst_port = readintbytesb(f, 2);
  udp_size = readintbytesb(f, 2);
  checksum = readintbytesb(f, 2);

  assert(pkt_size <= udp_size && pkt_size > 8);
  if (src_port == 53 || dst_port == 53) { // DNS
    dummy = parse_dns(f, pkt_size - 8);
  } else { // others
    if (valid(data)) {
      dummy = free_ec(data);
    }
    data_size = pkt_size - 8;
    data = malloc_ec(data_size);
    assert(valid(data));
    i = 0;
    while (i < data_size) {
      x = read_ec(f);
      assert(x >= 0);
      data[i] = x;
      i = i + 1;
    }
    print('\n');
  }
  return 0;
}

func parse_dns (f, data_size) {
  id = readintbytesb(f, 2);
  print(id);
  print('\n');
  flags = readintbytesb(f, 2);
  n_q = readintbytesb(f, 2);
  n_ans = readintbytesb(f, 2);
  n_auth = readintbytesb(f, 2);
  n_add = readintbytesb(f, 2);

  // questions
  i = 0;
  while (i < n_q) {
    stop = 0;
    first = 1;
    while (!stop) {
      len = read_ec(f);
      assert(len >= 0);
      if (len == 0) {
        stop = 1;
      } else {
        if (first) {
          first = 0;
        } else {
          print('.');
        }
        j = 0;
        while (j < len) {
          x = read_ec(f);
          assert(x >= 0);
          print(x);
          j = j + 1;
        }
      }
    }
    print('\n');
    type_q = readintbytesb(f, 2);
    class_q = readintbytesb(f, 2);
    i = i + 1;
  }

  // answers
  i = 0;
  while (i < n_ans) {
    ptr = readintbytesb(f, 2);
    assert(ptr == 49164);
    type_ans = readintbytesb(f, 2);
    class_ans = readintbytesb(f, 2);
    ttl = readintbytesb(f, 4);
    len = readintbytesb(f, 2);
    if (type_ans == 1) {
      assert(len == 4);
      addr = readintbytesb(f, 4);
      dummy = print_ip(addr);
    } else {
      assert(type_ans == 5 && len > 0);
      j = 0;
      while (j < len) {
        x = read_ec(f);
        assert(x >= 0);
        print(x);
        j = j + 1;
      }
    }
    print('\n');
    i = i + 1;
  }

  // authority
  // additional
  return 0;
}
```

```
220
221 main {
222   assert(valid(dst_mac) && valid(src_mac) && valid
          (data));
223   f = openb_ec("../inputs/pcap/bad.pcap", 1);
224   assert(valid(f));
225
226   x = read_ec(f);
227   assert(x == 212);
228   x = read_ec(f);
229   assert(x == 195);
230   x = read_ec(f);
231   assert(x == 178);
232   x = read_ec(f);
233   assert(x == 161);
234   maj_ver = readintbytesl(f, 2);
235   min_ver = readintbytesl(f, 2);
236   this_zone = readintbytesl(f, 4);
237   sigfigs = readintbytesl(f, 4);
238   snap_len = readintbytesl(f, 4); // number of
          packets
239   link_type = readintbytesl(f, 4);
240   assert(link_type == 1); // ethernet
241
242   inspectb (1, f, 12, 4, 0) {
243     ts_epoch = readintbytesl(f, 4);
244     ts_nanosec = readintbytesl(f, 4);
245     caplen = readintbytesl(f, 4);
246     length = readintbytesl(f, 4);
247     assert(caplen == length);
248
249     dummy = parse_ethernet(f, 0);
250   }
251
252   return 0;
253 }
```

C.7.3 Explicit Recovery Version

```
1  dst_mac = malloc_ec(6);
2  src_mac = malloc_ec(6);
3  dst_ip = 0;
4  src_ip = 0;
5  data = malloc_ec(1);
6
7  bad = 0;
8  idx = 0;
9  out = malloc_ec(1000);
10
11 func readintbytesl (f, n) { // little endian
12   x = read_ec(f);
13   if (x < 0) {
14     bad = 1;
15     return -1;
16   }
17   if (n == 1) {
18     return x;
19   } else {
20     y = readintbytesl(f, n-1);
21     return (y << 8) + x;
22   }
23 }
24
25 func readintbytesb (f, n) { // big endian
26   x = 0;
27   i = 0;
28   while (i < n) {
29     byte = read_ec(f);
30     if (byte < 0) {
31       bad = 1;
32       return -1;
33     }
34     x = x << 8;
35     x = x | byte;
36     i = i + 1;
37   }
38   return x;
39 }
40
41 func parse_ethernet (f, dummy) {
42   i = 0;
43   while (i < 6) {
44     x = read_ec(f);
45     if (x < 0) {
46       bad = 1;
47       return -1;
48     }
49     dst_mac[i] = x;
50     i = i + 1;
51   }
52   i = 0;
53   while (i < 6) {
54     x = read_ec(f);
55     if (x < 0) {
56       bad = 1;
57       return -1;
58     }
59     src_mac[i] = x;
60     i = i + 1;
61   }
62
63   ether_type = readintbytesl(f, 2);
64   if (bad || ether_type != 8) {
65     bad = 1;
66     return -1;
67   } // IPv4
68   dummy = parse_ipv4(f, 0);
69   return 0;
70 }
71
72 func check_sum (f, len) {
73   start = pos_ec(f);
74   if (len <= 0 || len % 2 != 0 || start < 0) {
75     bad = 1;
76     return -1;
77   }
78   sum = 0;
79   i = 0;
80   while (i < len / 2) {
81     x = readintbytesb(f, 2);
82     sum = sum + x;
83     i = i + 1;
84   }
85   hi = (sum >> 16) & 65535;
86   lo = sum & 65535;
87   x = seek_ec(f, start);
88   if (hi + lo != 65535 || x < 0) {
89     bad = 1;
90     return -1;
91   }
92   return 0;
93 }
94
95 func prtbuf (x) {
96   if (idx >= 1000) {
97     bad = 1;
98     return -1;
99   }
100  out[idx] = x;
101  idx = idx + 1;
102  return 0;
103 }
104
105 func prtbuf_ip (addr) {
106   dummy = prtbuf((addr >> 24) & 255);
107   dummy = prtbuf('.');
108   dummy = prtbuf((addr >> 16) & 255);
109   dummy = prtbuf('.');
110   dummy = prtbuf((addr >> 8) & 255);
111   dummy = prtbuf('.');
112   dummy = prtbuf(addr & 255);
113   return 0;
114 }
115
116 func parse_ipv4 (f, dummy) {
117   dummy = check_sum(f, 20); // no options
118   if (bad) {
119     return -1;
120   }
121
122   x = read_ec(f);
123   if (x < 0) {
```

```
    bad = 1;
    return -1;
  }
  version = (x & 240) >> 4;
  hdr_size = x & 15;
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  total = readintbytesb(f, 2);

  id = readintbytesb(f, 2);
  fragment = readintbytesb(f, 2);
  dont_frag = (fragment & 16384) >> 14;
  more_frag = (fragment & 8192) >> 13;
  offs_frag = fragment & 8191;

  ttl = read_ec(f);
  ip_type = read_ec(f);
  checksum = readintbytesb(f, 2);

  src_ip = readintbytesb(f, 4);
  dst_ip = readintbytesb(f, 4);

  if (bad || version != 4 || hdr_size != 5 ||
      total < hdr_size * 4 || fragment & 32768 !=
       0 || more_frag != 0 || offs_frag != 0 ||
      ttl < 0 || ip_type != 17) { // IPv4, no
      options, UDP
    bad = 1;
    return -1;
  }
  dummy = prtbuf('\n');
  dummy = prtbuf_ip(src_ip);
  dummy = prtbuf(' ');
  dummy = prtbuf_ip(dst_ip);
  dummy = prtbuf(' ');
  dummy = parse_udp(f, total - hdr_size * 4);
  return 0;
}

func parse_udp (f, pkt_size) {
  src_port = readintbytesb(f, 2);
  dst_port = readintbytesb(f, 2);
  udp_size = readintbytesb(f, 2);
  checksum = readintbytesb(f, 2);

  if (bad || pkt_size > udp_size || pkt_size <= 8)
       {
    bad = 1;
    return -1;
  }

  if (src_port == 53 || dst_port == 53) { // DNS
    dummy = parse_dns(f, pkt_size - 8);
  } else { // others
    if (valid(data)) {
      dummy = free_ec(data);
    }
    data_size = pkt_size - 8;
    data = malloc_ec(data_size);
    if (!valid(data)) {
      bad = 1;
      return -1;
    }
    i = 0;
    while (i < data_size) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
        return -1;
      }
      data[i] = x;
      i = i + 1;
    }
    dummy = prtbuf('\n');
  }
  return 0;
}

func parse_dns (f, data_size) {
  id = readintbytesb(f, 2);
  dummy = prtbuf(id);
  dummy = prtbuf('\n');
  flags = readintbytesb(f, 2);
  n_q = readintbytesb(f, 2);
  n_ans = readintbytesb(f, 2);
  n_auth = readintbytesb(f, 2);
  n_add = readintbytesb(f, 2);

  // questions
  i = 0;
  while (i < n_q) {
    stop = 0;
    first = 1;
    while (!stop) {
      len = read_ec(f);
      if (len < 0) {
        bad = 1;
        return -1;
      }
      if (len == 0) {
        stop = 1;
      } else {
        if (first) {
          first = 0;
        } else {
          dummy = prtbuf('.');
        }
        j = 0;
        while (j < len) {
          x = read_ec(f);
          if (x < 0 || idx >= 1000) {
            bad = 1;
            return -1;
          }
          out[idx] = x;
          j = j + 1;
          idx = idx + 1;
        }
      }
    }
    dummy = prtbuf('\n');
    type_q = readintbytesb(f, 2);
    class_q = readintbytesb(f, 2);
    i = i + 1;
  }

  // answers
  i = 0;
  while (i < n_ans) {
    ptr = readintbytesb(f, 2);
    if (ptr != 49164) {
      bad = 1;
      return -1;
    }
    type_ans = readintbytesb(f, 2);
    class_ans = readintbytesb(f, 2);
    ttl = readintbytesb(f, 4);
    len = readintbytesb(f, 2);
    if (type_ans == 1) {
      if (len != 4) {
        bad = 1;
        return -1;
      }
      addr = readintbytesb(f, 4);
      dummy = prtbuf_ip(addr);
    } else {
      if (type_ans != 5 || len <= 0) {
        bad = 1;
        return -1;
      }
      j = 0;
      while (j < len) {
        x = read_ec(f);
        if (x < 0 || idx >= 1000) {
          bad = 1;
          return -1;
        }
        out[idx] = x;
        j = j + 1;
```

```
          idx = idx + 1;
        }
      }
      dummy = prtbuf('\n');
      i = i + 1;
    }

    // authority
    // additional
    return 0;
  }

  main {
    if (!valid(dst_mac) || !valid(src_mac) || !valid
        (data) || !valid(out)) {
      exit(1);
    }
    f = openb_ec("../inputs/pcap/bad.pcap", 1);
    if (!valid(f)) {
      exit(1);
    }

    x = read_ec(f);
    if (x != 212) {
      exit(1);
    }
    x = read_ec(f);
    if (x != 195) {
      exit(1);
    }
    x = read_ec(f);
    if (x != 178) {
      exit(1);
    }
    x = read_ec(f);
    if (x != 161) {
      exit(1);
    }
    maj_ver = readintbytesl(f, 2);
    min_ver = readintbytesl(f, 2);
    this_zone = readintbytesl(f, 4);
    sigfigs = readintbytesl(f, 4);
    snap_len = readintbytesl(f, 4); // number of
        packets
    link_type = readintbytesl(f, 4);
    if (bad || link_type != 1) {
      exit(1);
    } // ethernet

    lookatb (1, f, 12, 4, 0) {
      bad = 0;
      idx = 0;
      ts_epoch = readintbytesl(f, 4);
      ts_nanosec = readintbytesl(f, 4);
      caplen = readintbytesl(f, 4);
      length = readintbytesl(f, 4);
      if (!bad && caplen == length) {
        dummy = parse_ethernet(f, 0);
        if (!bad) {
          i = 0;
          while (i < idx) {
            x = out[i];
            print(x);
            i = i + 1;
          }
        }
      } // skip unit
    }

    return 0;
  }
```

C.7.4 Conventional Version

```
  dst_mac = malloc_ec(6);
  src_mac = malloc_ec(6);
  dst_ip = 0;
  src_ip = 0;
  data = malloc_ec(1);

  bad = 0;
  idx = 0;
  out = malloc_ec(1000);

  func readintbytesl (f, n) { // little endian
    x = read_ec(f);
    if (x < 0) {
      bad = 1;
      return -1;
    }
    if (n == 1) {
      return x;
    } else {
      y = readintbytesl(f, n-1);
      return (y << 8) + x;
    }
  }

  func readintbytesb (f, n) { // big endian
    x = 0;
    i = 0;
    while (i < n) {
      byte = read_ec(f);
      if (byte < 0) {
        bad = 1;
        return -1;
      }
      x = x << 8;
      x = x | byte;
      i = i + 1;
    }
    return x;
  }

  func parse_ethernet (f, dummy) {
    i = 0;
    while (i < 6) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
        return -1;
      }
      dst_mac[i] = x;
      i = i + 1;
    }
    i = 0;
    while (i < 6) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
        return -1;
      }
      src_mac[i] = x;
      i = i + 1;
    }

    ether_type = readintbytesl(f, 2);
    if (bad || ether_type != 8) {
      bad = 1;
      return -1;
    } // IPv4
    dummy = parse_ipv4(f, 0);
    return 0;
  }

  func check_sum (f, len) {
    start = pos_ec(f);
    if (len <= 0 || len % 2 != 0 || start < 0) {
      bad = 1;
      return -1;
    }
    sum = 0;
    i = 0;
    while (i < len / 2) {
      x = readintbytesb(f, 2);
      sum = sum + x;
      i = i + 1;
    }
    hi = (sum >> 16) & 65535;
    lo = sum & 65535;
    x = seek_ec(f, start);
    if (hi + lo != 65535 || x < 0) {
```

```
    bad = 1;
    return -1;
  }
  return 0;
}

func prtbuf (x) {
  if (idx >= 1000) {
    bad = 1;
    return -1;
  }
  out[idx] = x;
  idx = idx + 1;
  return 0;
}

func prtbuf_ip (addr) {
  dummy = prtbuf((addr >> 24) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf((addr >> 16) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf((addr >> 8) & 255);
  dummy = prtbuf('.');
  dummy = prtbuf(addr & 255);
  return 0;
}

func parse_ipv4 (f, dummy) {
  dummy = check_sum(f, 20); // no options
  if (bad) {
    return -1;
  }

  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  version = (x & 240) >> 4;
  hdr_size = x & 15;
  x = read_ec(f);
  if (x < 0) {
    bad = 1;
    return -1;
  }
  total = readintbytesb(f, 2);

  id = readintbytesb(f, 2);
  fragment = readintbytesb(f, 2);
  dont_frag = (fragment & 16384) >> 14;
  more_frag = (fragment & 8192) >> 13;
  offs_frag = fragment & 8191;

  ttl = read_ec(f);
  ip_type = read_ec(f);
  checksum = readintbytesb(f, 2);

  src_ip = readintbytesb(f, 4);
  dst_ip = readintbytesb(f, 4);

  if (bad || version != 4 || hdr_size != 5 ||
      total < hdr_size * 4 || fragment & 32768 !=
       0 || more_frag != 0 || offs_frag != 0 ||
      ttl < 0 || ip_type != 17) { // IPv4, no
      options, UDP
    bad = 1;
    return -1;
  }
  dummy = prtbuf('\n');
  dummy = prtbuf_ip(src_ip);
  dummy = prtbuf(' ');
  dummy = prtbuf_ip(dst_ip);
  dummy = prtbuf(' ');
  dummy = parse_udp(f, total - hdr_size * 4);
  return 0;
}

func parse_udp (f, pkt_size) {
  src_port = readintbytesb(f, 2);
  dst_port = readintbytesb(f, 2);
  udp_size = readintbytesb(f, 2);
  checksum = readintbytesb(f, 2);

  if (bad || pkt_size > udp_size || pkt_size <= 8)
      {
    bad = 1;
    return -1;
  }

  if (src_port == 53 || dst_port == 53) { // DNS
    dummy = parse_dns(f, pkt_size - 8);
  } else { // others
    if (valid(data)) {
      dummy = free_ec(data);
    }
    data_size = pkt_size - 8;
    data = malloc_ec(data_size);
    if (!valid(data)) {
      bad = 1;
      return -1;
    }
    i = 0;
    while (i < data_size) {
      x = read_ec(f);
      if (x < 0) {
        bad = 1;
        return -1;
      }
      data[i] = x;
      i = i + 1;
    }
    dummy = prtbuf('\n');
  }
  return 0;
}

func parse_dns (f, data_size) {
  id = readintbytesb(f, 2);
  dummy = prtbuf(id);
  dummy = prtbuf('\n');
  flags = readintbytesb(f, 2);
  n_q = readintbytesb(f, 2);
  n_ans = readintbytesb(f, 2);
  n_auth = readintbytesb(f, 2);
  n_add = readintbytesb(f, 2);

  // questions
  i = 0;
  while (i < n_q) {
    stop = 0;
    first = 1;
    while (!stop) {
      len = read_ec(f);
      if (len < 0) {
        bad = 1;
        return -1;
      }
      if (len == 0) {
        stop = 1;
      } else {
        if (first) {
          first = 0;
        } else {
          dummy = prtbuf('.');
        }
        j = 0;
        while (j < len) {
          x = read_ec(f);
          if (x < 0 || idx >= 1000) {
            bad = 1;
            return -1;
          }
          out[idx] = x;
          j = j + 1;
          idx = idx + 1;
        }
      }
    }
    dummy = prtbuf('\n');
    type_q = readintbytesb(f, 2);
    class_q = readintbytesb(f, 2);
    i = i + 1;
```

```
246     }
247
248     // answers
249     i = 0;
250     while (i < n_ans) {
251       ptr = readintbytesb(f, 2);
252       if (ptr != 49164) {
253         bad = 1;
254         return -1;
255       }
256       type_ans = readintbytesb(f, 2);
257       class_ans = readintbytesb(f, 2);
258       ttl = readintbytesb(f, 4);
259       len = readintbytesb(f, 2);
260       if (type_ans == 1) {
261         if (len != 4) {
262           bad = 1;
263           return -1;
264         }
265         addr = readintbytesb(f, 4);
266         dummy = prtbuf_ip(addr);
267       } else {
268         if (type_ans != 5 || len <= 0) {
269           bad = 1;
270           return -1;
271         }
272         j = 0;
273         while (j < len) {
274           x = read_ec(f);
275           if (x < 0 || idx >= 1000) {
276             bad = 1;
277             return -1;
278           }
279           out[idx] = x;
280           j = j + 1;
281           idx = idx + 1;
282         }
283       }
284       dummy = prtbuf('\n');
285       i = i + 1;
286     }
287
288     // authority
289     // additional
290     return 0;
291   }
292
293   main {
294     if (!valid(dst_mac) || !valid(src_mac) || !valid
          (data) || !valid(out)) {
295       exit(1);
296     }
297     f = openb_ec("../inputs/pcap/bad.pcap", 1);
298     if (!valid(f)) {
299       exit(1);
300     }
301
302     x = read_ec(f);
303     if (x != 212) {
304       exit(1);
305     }
306     x = read_ec(f);
307     if (x != 195) {
308       exit(1);
309     }
310     x = read_ec(f);
311     if (x != 178) {
312       exit(1);
313     }
314     x = read_ec(f);
315     if (x != 161) {
316       exit(1);
317     }
318     maj_ver = readintbytesl(f, 2);
319     min_ver = readintbytesl(f, 2);
320     this_zone = readintbytesl(f, 4);
321     sigfigs = readintbytesl(f, 4);
322     snap_len = readintbytesl(f, 4); // number of
          packets
323     link_type = readintbytesl(f, 4);
324     if (bad || link_type != 1) {
325       exit(1);
326     } // ethernet
327
328     finish = 0;
329     while (!finish) {
330       bad = 0;
331       idx = 0;
332       ts_epoch = readintbytesl(f, 4);
333       ts_nanosec = readintbytesl(f, 4);
334       caplen = readintbytesl(f, 4);
335       length = readintbytesl(f, 4);
336       if (!bad && length >= 0) {
337         eou = pos_ec(f) + length;
338         if (!bad && caplen == length) {
339           dummy = parse_ethernet(f, 0);
340           if (pos_ec(f) < 0 || pos_ec(f) > eou) {
341             bad = 1;
342           }
343           if (!bad) {
344             i = 0;
345             while (i < idx) {
346               x = out[i];
347               print(x);
348               i = i + 1;
349             }
350           }
351         } // skip unit
352         x = seek_ec(f, eou);
353         if (x < 0) {
354           finish = 1;
355         }
356       } else { // give up
357         finish = 1;
358       }
359     }
360
361     return 0;
362   }
```

D. Extending RIFL for Handling Errors Explicitly

For evaluation, we extend RIFL to support handling errors explicitly. In general, however, we recommend developers to use only the core language features that handle errors implicitly.

We extend RIFL to support look-at loops. A look-at loop iterates though the inputs as an inspect loop does, except that the look-at loop does not attempt to detect or recover from errors. Figures 21 and 22 present the semantics for look-at loops for text input files, using the same helper functions as for Figures 12 and 13. Figures 23 and 24 present the semantics for look-at loops for binary input files, using the same helper functions as for Figures 14 and 15.

We also extend RIFL to support an additional set of system calls for file and memory operations. These extensions resemble the core RIFL system calls when there are no errors, but return explicit error codes instead of triggering error recovery when there are errors. Table 11 presents the error-code extensions for core RIFL system calls. The two columns present the system calls in the core RIFL language (Core-language interface) and in the extended language (Error-code extension), respectively. Each row contains the two versions of a system call. Figures 25–28 present the semantics for these error-code system calls.

To support these error-code extensions, we extend RIFL with the `valid` predicate that tests whether a file variable is not null. Figure 29 presents the semantics for the `valid`

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{false}}{\langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(lookt-end)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \\ \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle \rangle \quad dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle \rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') \geq n' \vee \gamma'(\Lambda'(l')) \in osd' \end{array}}{\langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle str, \Lambda'(l'), sou, \langle eou, eos, osd \rangle \rangle], \sigma'_D \rangle, \text{ok} \rangle} \quad \text{(lookt-osd)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \\ \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle \rangle \quad dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle \rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in eos' \end{array}}{\langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle str, \Lambda'(l') + 1, sou, \langle eou, eos, osd \rangle \rangle], \sigma'_D \rangle, \text{ok} \rangle} \quad \text{(lookt-eos)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \\ \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle \rangle \quad dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \sigma'_F(\sigma'_S(f)) = \langle str, l', sou', \langle eou', eos', osd' \rangle \rangle \quad \sigma'_D(str) = \langle \gamma', n' \rangle \quad \Lambda'(l') < n' \quad \gamma'(\Lambda'(l')) \in eou' \\ \langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle str, \Lambda'(l') + 1, sou, \langle eou, eos, osd \rangle \rangle], \sigma'_D \rangle \rangle \Downarrow_{\text{s}} \xi \end{array}}{\langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \xi} \quad \text{(lookt-prgr)}$$

Figure 21: Semantics for iterators without error handling—with good text inputs

$$\frac{\sigma_S(f) = \text{null} \vee \sigma_S(d_u) = \text{null} \vee \sigma_S(d_s) = \text{null}}{\langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(lookt-null)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{err}}{\langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(lookt-bad)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) \neq \varnothing}{\langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{bad} \rangle} \quad \text{(lookt-dupl)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma_S(d_u) \neq \text{null} \quad \sigma_S(d_s) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{true} \quad \Omega(d_u) \cap \Omega(d_s) = \varnothing \\ \sigma_F(\sigma_S(f)) = \langle str, l, sou, \langle eou, eos, osd \rangle \rangle \quad dlm = \langle \Omega(d_u), \Omega(d_s), (eou \cup eos \cup osd) \setminus (\Omega(d_u) \cup \Omega(d_s)) \rangle \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, dlm \rangle], \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \sigma', \text{bad} \rangle \end{array}}{\langle \texttt{lookatt}(e, f, d_u, d_s)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \sigma', \text{bad} \rangle} \quad \text{(lookt-abort)}$$

Figure 22: Semantics for iterators without error handling—with bad text inputs

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{e}} \text{false}}{\langle \texttt{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(lookb-end)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma = \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \quad \langle e, \sigma \rangle \Downarrow_{\text{e}} \text{true} \quad \langle o, \sigma \rangle \Downarrow_{\text{e}} u_o \quad \langle w, \sigma \rangle \Downarrow_{\text{e}} u_w \quad \langle c, \sigma \rangle \Downarrow_{\text{e}} u_c \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, cut \rangle \\ \sigma_D(str) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \quad parseint(l + u_o, u_w) = v \quad cut = l \vee u_o = u_w = u_c = 0 \vee v < 0 \end{array}}{\langle \texttt{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, cut, sou, cut \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad \text{(lookb-eos)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma = \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \quad \langle e, \sigma \rangle \Downarrow_{\text{e}} \text{true} \quad \langle o, \sigma \rangle \Downarrow_{\text{e}} u_o \quad \langle w, \sigma \rangle \Downarrow_{\text{e}} u_w \quad \langle c, \sigma \rangle \Downarrow_{\text{e}} u_c \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, cut \rangle \\ \sigma_D(str) = \langle \gamma, n \rangle \quad u_o \geq 0 \quad u_w \geq 0 \quad parseint(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = cut' \quad l < cut' \leq cut \\ \langle s, \langle \sigma_S, \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l, l, cut' \rangle], \sigma_D \rangle \rangle \Downarrow_{\text{s}} \langle \langle \sigma'_S, \sigma'_H, \sigma'_F, \sigma'_D \rangle, \text{ok} \rangle \\ \langle \texttt{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma'_S, \sigma'_H, \sigma'_F[\sigma'_S(f) \mapsto \langle str, cut', sou, cut \rangle], \sigma'_D \rangle \rangle \Downarrow_{\text{s}} \xi \end{array}}{\langle \texttt{lookatb}(e, f, o, w, c)\{s\}, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle \rangle \Downarrow_{\text{s}} \xi} \quad \text{(lookb-prgr)}$$

Figure 23: Semantics for iterators without error handling—with good binary inputs

$$\frac{\sigma_S(f) = \text{null}}{\langle \texttt{lookatb}(e,f,d_u,d_s)\{s\}, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \text{bad}\rangle} \quad \text{(lookb-null)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \quad \langle e,\sigma\rangle \Downarrow_\text{e} \text{err} \vee \langle o,\sigma\rangle \Downarrow_\text{e} \text{err} \vee \langle w,\sigma\rangle \Downarrow_\text{e} \text{err} \vee \langle c,\sigma\rangle \Downarrow_\text{e} \text{err}}{\langle \texttt{lookatb}(e,f,o,w,c)\{s\}, \sigma\rangle \Downarrow_\text{s} \langle \sigma, \text{bad}\rangle} \quad \text{(lookb-bad)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \quad \langle e,\sigma\rangle \Downarrow_\text{e} \text{true} \quad \langle o,\sigma\rangle \Downarrow_\text{e} u_o \quad \langle w,\sigma\rangle \Downarrow_\text{e} u_w \quad \langle c,\sigma\rangle \Downarrow_\text{e} u_c \\ \sigma_F(\sigma_S(f)) = \langle str, l, sou, cut\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad u_o < 0 \vee u_w < 0 \vee u_c < 0\end{array}}{\langle \texttt{lookatb}(e,f,o,w,c)\{s\}, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \text{bad}\rangle} \quad \text{(lookb-neg)}$$

$$\frac{\begin{array}{c}\sigma_S(f) \neq \text{null} \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \quad \langle e,\sigma\rangle \Downarrow_\text{e} \text{true} \quad \langle o,\sigma\rangle \Downarrow_\text{e} u_o \quad \langle w,\sigma\rangle \Downarrow_\text{e} u_w \quad \langle c,\sigma\rangle \Downarrow_\text{e} u_c \\ \sigma_F(\sigma_S(f)) = \langle str, l, sou, cut\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad u_o \geq 0 \quad u_w \geq 0 \\ parseint(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = cut' \quad cut' > cut\end{array}}{\langle \texttt{lookatb}(e,f,o,w,c)\{s\}, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle, \text{bad}\rangle} \quad \text{(lookb-large)}$$

$$\frac{\begin{array}{c}\sigma_S(F) \neq \text{null} \quad \sigma = \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle \quad \langle e,\sigma\rangle \Downarrow_\text{e} \text{true} \quad \langle o,\sigma\rangle \Downarrow_\text{e} u_o \quad \langle w,\sigma\rangle \Downarrow_\text{e} u_w \quad \langle c,\sigma\rangle \Downarrow_\text{e} u_c \\ \sigma_F(\sigma_S(f)) = \langle str, l, sou, cut\rangle \quad \sigma_D(str) = \langle \gamma, n\rangle \quad u_o \geq 0 \quad u_w \geq 0 \\ parseint(l + u_o, u_w) = v \quad v \geq 0 \quad u_c \geq 0 \quad l + u_o + u_w + v + u_c = cut' \quad l < cut' \leq cut \\ \langle s, \langle \sigma_S,\sigma_H,\sigma_F[\sigma_S(f) \mapsto \langle str, l, l, cut'\rangle], \sigma_D\rangle\rangle \Downarrow_\text{s} \langle \sigma', \text{bad}\rangle\end{array}}{\langle \texttt{lookatb}(e,f,o,w,c)\{s\}, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle \sigma', \text{bad}\rangle} \quad \text{(lookb-abort)}$$

Figure 24: Semantics for iterators without error handling—with bad binary inputs

$$\frac{\langle e, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{e} \text{err}}{\langle a = \texttt{malloc\_ec}(e), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[a \mapsto \text{null}],\sigma_H,\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(malloc-ec-bad)}$$

$$\frac{\langle e, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{e} u \quad u \leq 0}{\langle a = \texttt{malloc\_ec}(e), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[a \mapsto \text{null}],\sigma_H,\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(malloc-ec-neg)}$$

$$\frac{\langle e, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{e} u \quad u > 0 \quad \text{heap allocate}(u) = \bot}{\langle a = \texttt{malloc\_ec}(e), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[a \mapsto \text{null}],\sigma_H,\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(malloc-ec-ovf)}$$

$$\frac{\langle e, \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{e} u \quad u > 0 \quad \text{heap allocate}(u) = \text{addr}}{\langle a = \texttt{malloc\_ec}(e), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[a \mapsto \text{addr}],\sigma_H[\text{addr} \mapsto \langle [0 \mapsto 0, 1 \mapsto 0, \ldots, u-1 \mapsto 0], u\rangle],\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(malloc-ec-ok)}$$

$$\frac{\sigma_S(a) = \text{null}}{\langle x = \texttt{free\_ec}(a), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[x \mapsto -1],\sigma_H,\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(free-ec-null)}$$

$$\frac{\sigma_S(a) \neq \text{null}}{\langle x = \texttt{free\_ec}(a), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[x \mapsto 0, a \mapsto \text{null}],\sigma_H[\sigma_S(a) \mapsto \bot],\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(free-ec-ok)}$$

Figure 25: Semantics for arrays with error codes

$$\frac{\text{file } str \text{ does not exist}}{\langle f = \texttt{opent\_ec}(str), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[f \mapsto \text{null}],\sigma_H,\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(opent-ec-bad)}$$

$$\frac{\text{file } str \text{ exists}}{\langle f = \texttt{opent\_ec}(str), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[f \mapsto \text{hndl}],\sigma_H,\sigma_F[\text{hndl} \mapsto \langle str, 0, 0, \langle \varnothing, \varnothing, \varnothing\rangle\rangle],\sigma_D\rangle, \text{ok}\rangle} \quad \text{(opent-ec-ok)}$$

$$\frac{\text{file } str \text{ does not exist}}{\langle f = \texttt{openb\_ec}(str), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[f \mapsto \text{null}],\sigma_H,\sigma_F,\sigma_D\rangle, \text{ok}\rangle} \quad \text{(openb-ec-bad)}$$

$$\frac{\text{file } str \text{ exists} \quad \sigma_D(str) = \langle \gamma, n\rangle}{\langle f = \texttt{openb\_ec}(str), \langle \sigma_S,\sigma_H,\sigma_F,\sigma_D\rangle\rangle \Downarrow_\text{s} \langle\langle \sigma_S[f \mapsto \text{hndl}],\sigma_H,\sigma_F[\text{hndl} \mapsto \langle str, 0, 0, n\rangle],\sigma_D\rangle, \text{ok}\rangle} \quad \text{(openb-ec-ok)}$$

Figure 27: Semantics for opening input files with error codes

$$\frac{\sigma_S(f) = \text{null}}{\langle x = \texttt{seek\_ec}(f,e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(sk-ec-null)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e \text{err}}{\langle x = \texttt{seek\_ec}(f,e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(sk-ec-bad)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad u < sou}{\langle x = \texttt{seek\_ec}(f,e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(sk-ec-l)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad sou \leq u \leq \Lambda(l)}{\langle x = \texttt{seek\_ec}(f,e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto u], \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, u, sou, ud \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad \text{(sk-ec-ok)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \langle e, \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e u \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad u > \Lambda(l)}{\langle x = \texttt{seek\_ec}(f,e), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(sk-ec-r)}$$

$$\frac{\sigma_S(f) = \text{null}}{\langle x = \texttt{read\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(frd-ec-null)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad l = \Lambda(l)}{\langle x = \texttt{read\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto -1], \sigma_H, \sigma_F, \sigma_D \rangle, \text{ok} \rangle} \quad \text{(frd-ec-out)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad l < \Lambda(l)}{\langle x = \texttt{read\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_s \langle \langle \sigma_S[x \mapsto \gamma(l)], \sigma_H, \sigma_F[\sigma_S(f) \mapsto \langle str, l+1, ud \rangle], \sigma_D \rangle, \text{ok} \rangle} \quad \text{(frd-ec-ok)}$$

Figure 28: Semantics for accessing input files with error codes

$$\frac{\sigma_S(f) = \text{null}}{\langle \texttt{end\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e -1} \quad \text{(end-ec-null)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad l = \Lambda(l)}{\langle \texttt{end\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e \text{true}} \quad \text{(end-ec-t)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle \quad \sigma_D(str) = \langle \gamma, n \rangle \quad l < \Lambda(l)}{\langle \texttt{end\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e \text{false}} \quad \text{(end-ec-f)}$$

$$\frac{\sigma_S(f) = \text{null}}{\langle \texttt{pos\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e -1} \quad \text{(pos-ec-null)}$$

$$\frac{\sigma_S(f) \neq \text{null} \quad \sigma_F(\sigma_S(f)) = \langle str, l, sou, ud \rangle}{\langle \texttt{pos\_ec}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e l} \quad \text{(pos-ec-ok)}$$

Figure 26: Semantics for input file expressions with error codes

$$\frac{\sigma_S(f) \neq \text{null}}{\langle \texttt{valid}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e \text{true}} \quad \text{(fvalid-t)}$$

$$\frac{\sigma_S(f) = \text{null}}{\langle \texttt{valid}(f), \langle \sigma_S, \sigma_H, \sigma_F, \sigma_D \rangle\rangle \Downarrow_e \text{false}} \quad \text{(fvalid-f)}$$

Figure 29: Semantics for expressions to support error-code extensions

predicate. In addition, the rules for `inspectt`, `inspectb`, `end`, `pos`, `seek`, and `read` should also consider the new situation where $\sigma_S(f) = \text{null}$.

Table 11: Error-code interfaces for system calls

Core-language interface	Error-code extension
$a = \texttt{malloc}(e)$	$a = \texttt{malloc\_ec}(e)$
$\texttt{free}(a)$	$x = \texttt{free\_ec}(a)$
$\texttt{end}(f)$	$\texttt{end\_ec}(f)$
$\texttt{pos}(f)$	$\texttt{pos\_ec}(f)$
$f = \texttt{opent}(str)$	$f = \texttt{opent\_ec}(str)$
$f = \texttt{openb}(str)$	$f = \texttt{openb\_ec}(str)$
$\texttt{seek}(f, e)$	$x = \texttt{seek\_ec}(f, e)$
$x = \texttt{read}(f)$	$x = \texttt{read\_ec}(f)$

References

[1] CVE – common vulnerabilities and exposures. http://cve.mitre.org/.

[2] Boost C++ libraries. http://www.boost.org/.

[3] Graphics interchange format version 89a. http://www.w3.org/Graphics/GIF/spec-gif89a.txt.

[4] Wavefront OBJ file format summary. http://www.fileformat.info/format/wavefrontobj/egff.htm.

[5] Python 3.5.0 documentation. https://docs.python.org/3/.

[6] Wireshark. https://www.wireshark.org/.

[7] T. Anderson and R. Kerr. Recovery blocks in action: A system supporting high reliability. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 447–457, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[8] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. Softw. Eng.*, 11(12):1491–1501, Dec. 1985. ISSN 0098-5589. doi: 10.1109/TSE.1985.231893.

[9] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM. doi: 10.1145/1455567.1455599.

[10] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 387–401, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3168-7. doi: 10.1109/SP.2008.22.

[11] T. Boutell. PNG (portable network graphics) specification version 1.0, 1997.

[12] T. Bray. Javascript object notation (JSON) data interchange format. http://www.rfc-editor.org/rfc/rfc7159.txt, Mar. 2014. RFC 7159.

[13] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 311–325, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2819-8. doi: 10.1109/CSF.2007.17.

[14] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. Detecting and escaping infinite loops with Jolt. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 609–633, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-22654-0.

[15] K. Chandy and C. Ramamoorthy. Rollback and recovery strategies for computer programs. *Computers, IEEE Transactions on*, C-21(6):546–556, June 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009007.

[16] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. L. Erlbaum Associates, 1988. ISBN 9780805802832.

[17] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 133–147, New York, NY, USA, 2005. ACM. ISBN 1-59593-079-5. doi: 10.1145/1095810.1095824.

[18] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado. Bouncer: Securing software by blocking bad input. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 117–130, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294274.

[19] F. Cristian. Exception handling and software fault tolerance. *IEEE Trans. Comput.*, 31(6):531–540, June 1982. ISSN 0018-9340. doi: 10.1109/TC.1982.1676035.

[20] O.-J. Dahl and K. Nygaard. SIMULA: An ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, Sept. 1966. ISSN 0001-0782. doi: 10.1145/365813.365819.

[21] J. Darley and B. Latane. Bystander intervention in emergencies: Diffusion of responsibility. *Journal of Personality and Social Psychology*, pages 377–383, Aug. 1968.

[22] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492.

[23] L. Degioanni, F. Risso, and G. Varenni. PCAP next generation dump file format. https://www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html, Mar. 2004.

[24] B. Demsky and A. Dash. Bristlecone: A language for robust software systems. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 490–515, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-70591-8. doi: 10.1007/978-3-540-70592-5_21.

[25] B. Demsky and M. C. Rinard. Goal-directed reasoning for specification-based data structure repair. *IEEE Trans. Softw. Eng.*, 32(12):931–951, Dec. 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.122.

[26] B. Demsky and S. Sundaramurthy. Bristlecone: Language support for robust software applications. *IEEE Trans. Softw. Eng.*, 37(1):4–23, Jan. 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.27.

[27] B. Demsky, J. Zhou, and W. Montaz. Recovery tasks: An automated approach to failure recovery. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 229–244, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16611-3, 978-3-642-16611-2.

[28] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 162–171, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: 10.1145/1134285.1134309.

[29] M. Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84–, Mar. 1997. ISSN 0163-5948. doi: 10.1145/251880.251992.

[30] T. Dybå, V. B. Kampenes, and D. I. Sjøberg. A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8):745 – 755, 2006. ISSN 0950-5849.

[31] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, Nov. 1976. ISSN 0001-0782. doi: 10.1145/360363.360369.

[32] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

[33] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, Dec. 1975. ISSN 0001-0782. doi: 10.1145/361227.361230.

[34] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013. ISBN 0133260224, 9780133260229.

[35] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992. ISBN 1558601902.

[36] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983. ISSN 0360-0300. doi: 10.1145/289.291.

[37] E. Hamilton. JPEG file interchange format, 1992.

[38] S. G. Hart and L. E. Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In P. A. Hancock and N. Meshkati, editors, *Human Mental Workload*, volume 52 of *Advances in Psychology*, pages 139 – 183. North-Holland, 1988.

[39] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164.

[40] K. Huang, J. Wu, and E. B. Fernández. A generalized forward recovery checkpointing scheme. In *IPPS/SPDP Workshops*, pages 623–643, 1998. doi: 10.1007/3-540-64359-1_732.

[41] K. E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5.

[42] B. Latane and J. Darley. Group inhibition of bystander intervention in emergencies. *Journal of Personality and Social Psychology*, pages 215–221, Oct. 1968.

[43] B. Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, Mar. 1988. ISSN 0001-0782. doi: 10.1145/42392.42399.

[44] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM*, 20(8):564–576, Aug. 1977. ISSN 0001-0782. doi: 10.1145/359763.359789.

[45] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 128–137, New York, NY, USA, 1977. ACM. doi: 10.1145/800022.808319.

[46] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. Automatic input rectification. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 80–90, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

[47] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 439–452, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535888.

[48] F. Long, S. Sidiroglou-Douskos, and M. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 227–238, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594337.

[49] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, Dec. 1990. ISSN 0001-0782. doi: 10.1145/96267.96279.

[50] C. Moler and U. of New Mexico. Department of Computer Science. *MATLAB User's Guide: November 1980*. Technical report (University of New Mexico. Department of Computer Science). Department of Computer Science, College of Engineering, University of New Mexico, 1981.

[51] H. H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, pages 15–30, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-893-0. doi: 10.1145/1296907.1296912.

[52] PKWARE. .ZIP application note. https://www.pkware.com/support/zip-app-note/.

[53] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. *SIGOPS Oper. Syst. Rev.*, 39(5):235–248, Oct. 2005. ISSN 0163-5980. doi: 10.1145/1095809.1095833.

[54] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015. URL https://www.R-project.org.

[55] G. Radin. The early history and characteristics of pl/i. *SIGPLAN Not.*, 13(8):227–241, Aug. 1978. ISSN 0362-1340. doi: 10.1145/960118.808389.

[56] M. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 324–334, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8. doi: 10.1145/1183401.1183447.

[57] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association.

[58] M. C. Rinard. Living in the comfort zone. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 611–622, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297072.

[59] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[60] J. E. Sammet. Basic elements of COBOL 61. *Commun. ACM*, 5(5):237–253, May 1962. ISSN 0001-0782. doi: 10.1145/367710.367721.

[61] T. Scholte, D. Balzarotti, and E. Kirda. Quo vadis? a study of the evolution of input validation vulnerabilities in web applications. In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, FC'11, pages 284–298, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-27575-3. doi: 10.1007/978-3-642-27576-0_24.

[62] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda. An empirical analysis of input validation mechanisms in web applications and languages. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1419–1426, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. doi: 10.1145/2245276.2232004.

[63] Y. Shafranovich. Common format and mime type for comma-separated values (CSV) files. https://tools.ietf.org/html/rfc4180, Oct. 2005. RFC 4180.

[64] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM. ISBN 0-89791-710-3. doi: 10.1145/224964.224987.

[65] S. Sidiroglou and A. D. Keromytis. Using Execution Transactions To Recover From Buffer Overflow Attacks. Technical report, Columbia University Computer Science Department, 2004. URL http://academiccommons.columbia.edu/item/ac:109823. CUCS–031–04.

[66] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: Automatic software self-healing using rescue points. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 37–48, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508250.

[67] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025133.

[68] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 43–54, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737988.

[69] S. Siegel. *Nonparametric statistics for the behavioral sciences*. McGraw-Hill series in psychology. McGraw-Hill, 1956.

[70] G. L. Steele, Jr. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

[71] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the oklahoma update. *IEEE Parallel Distrib. Technol.*, 1(4):58–71, Nov. 1993. ISSN 1063-6552. doi: 10.1109/88.260295.

[72] Symantec Inc. Symantec internet security threat report: Vol. VII. Technical report, Mar. 2005.

[73] J. Tucek, J. Newsome, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song. Sweeper: A lightweight end-to-end system for defending against fast worms. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 115–128, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-636-3. doi: 10.1145/1272996.1273010.

[74] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM. ISBN 0-89791-247-0. doi: 10.1145/38765.38828.

[75] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 193–204, New York, NY, USA, 2004. ACM. ISBN 1-58113-862-8. doi: 10.1145/1015467.1015489.

[76] A. v. Wijngaarden. *Report on the Algorithmic Language ALGOL 68*. Printing by the Mathematisch Centrum, 1969.

ISBN B0007IUUXM.

[77] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945. ISSN 00994987. doi: 10.2307/3001968.

[78] N. Wirth. *Programming in Modula-2*. Springer-Verlag TELOS, Santa Clara, CA, USA, 1983. ISBN 555038521X.

[79] N. Wirth. Recollections about the development of Pascal. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 333–342, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. doi: 10.1145/154766.155378.

[80] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.

[81] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.