Robust Programs with Filtered Iterators

Jiasi Shen
MIT EECS and CSAIL
Cambridge, MA, USA
jiasi@csail.mit.edu

Martin Rinard
MIT EECS and CSAIL
Cambridge, MA, USA
rinard@csail.mit.edu

Abstract

We present a new language construct, *filtered iterators*, for robust input processing. Filtered iterators are designed to eliminate many common input processing errors while enabling robust continued execution. The design is inspired by (1) observed common input processing errors and (2) successful strategies implemented by human developers fixing input processing errors. Filtered iterators decompose inputs into input units and atomically and automatically discard units that trigger errors. Statistically significant results from a developer study demonstrate the effectiveness of filtered iterators in enabling developers to produce robust input processing code without common input processing defects.

CCS Concepts • **Software and its engineering → Control structures**; **Error handling and recovery**; • **Security and privacy →** *Software and application security*;

Keywords Input processing; robustness; filtered iterators; atomicity; continued execution; controlled experiment

ACM Reference Format:
Jiasi Shen and Martin Rinard. 2017. Robust Programs with Filtered Iterators. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136030

1 Introduction

We present the design and evaluation of a new programming language construct, *filtered iterators*, for robust input processing. Filtered iterators target defects and security vulnerabilities in input processing code and provide continued successful execution in the face of otherwise fatal or exploitable errors.

The design of filtered iterators was driven by observed strategies from human developers fixing input processing defects [27]. These strategies exploit a structure present in many applications: the application breaks the input into *input units*, then processes each input unit in sequence. The defect involves unanticipated corner case input units that trigger errors. Developers often fix such defects by modifying the program to recognize and discard such corner case input units (they are often malformed and should be discarded even by a correct program) [27]. Filtered iterators automate this strategy to provide the following benefits:

- **Survival:** Filtered iterators eliminate program crashes and enable successful continued execution, automatically and atomically discarding input units that trigger fatal errors, cause assertion failures, 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 potentially omit any check—and the associated error-handling code—that is otherwise required to protect the program from crashes and security exploits.

We empirically evaluate the effectiveness of filtered iterators in a study where 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, smaller cyclomatic complexity, and fewer conditional clauses. Moreover, the use of filtered iterators, by construction, completely eliminates many common sources of defects and vulnerabilities. Filtered iterators therefore address the following problem statement: *"Design and evaluate a new programming language construct that eliminates input processing errors to enhance survival, robustness, reliability, and security."*

1.1 Main Concept

Filtered iterators first split the input into input units, then iterate over the input units. Example input units include lines in a text file, rows in a database table, packets in network traffic, and images in an animation. Each iteration executes sequentially as an atomic (all-or-nothing) transaction that processes one input unit: if the iteration succeeds, it executes normally; if the iteration encounters a runtime error, it is rolled back and discarded atomically. The implementation of atomicity involves executing and rolling back the program state on errors, using standard atomic execution techniques that checkpoint program state and delay externally visible

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136030

outputs until processing commits (see Section 3.2). Apart from ensuring the atomic processing of each input unit, filtered iterators also discard *bad* input units—input units that trigger errors.[1] These errors include execution integrity errors, such as memory errors and arithmetic errors, as well as application-specific errors caught by failed assertions. In short, filtered iterators implement the following schema:

```
split input into input units
iterate over input units {
  atomic transaction {
    delay outputs until commit
    process input unit
    if unhandled exception or assertion failure {
      abort transaction
    } else{
      commit transaction
      release outputs
} } }
```

The synergy between atomic error recovery and discarding bad input units enables the program to continue processing the remaining input units, as if the bad input units had never existed.[2] For example, a program that uses a conventional language to extract files from a ZIP [1] archive may crash on files that trigger unanticipated conditions. Using filtered iterators, the new program will (1) survive, (2) extract all the files that the program can successfully extract, and (3) automatically skip the files that the program cannot process. Filtered iterators, therefore, use the program itself to find and filter bad input units.

Relevance: Input processing defects occur frequently in practice and often have serious consequences such as program failures, data corruption, and security vulnerabilities [4, 26, 27, 30, 45, 50]. The consequences can be especially severe for programs that are exposed to potentially malicious inputs that purposefully target defects that well-formed or anticipated inputs do not expose. Filtered iterators address these problems.

The design of filtered iterators is based on two empirical observations: (1) many of these defects correspond to uncommon corner cases that developers simply overlook and (2) when developers fix these defects, they often maximize program utility via robust continued execution. Specifically, many human bug fixes for null dereferences and divide-by-zero errors do not throw an exception or exit the program. They instead skip the computation associated with the current input unit and continue to process subsequent input units [27]. For example, in the CVE database [4], fixes to errors CVE-2013-2483, CVE-2012-4286, CVE-2012-4285, CVE-2012-1143, CVE-2012-4507, and CVE-2011-4153 all exhibit this pattern. Many of these bug fixes are specifically designed to enable the program to provide useful partial results in the face of bad input units [27]. Filtered iterators automate this empirical bug fixing strategy that developers repeatedly apply in practice.

Applicability in Different Contexts: Filtered iterators are designed for application contexts where obtaining partial results computed from a subset of the input units is preferable to simply terminating when the computation encounters a bad input unit. This property may be particularly relevant when the input contains multiple largely-independent input units and when the application can deliver acceptable results even after some input units are discarded [27, 38]. Example application contexts include: servers that process requests [18], big data analytics that process rows in large tables [29], video decoders that process video frames [52], network routers that process packets [24], embedded systems that process events [47], and text processors that process files line by line. This paper investigates applications that process text files and images (see Sections 2, 3.2, and 5). All of these applications benefit from continued execution in the face of errors and can use filtered iterators to process input units that can be correctly handled, skipping bad input units that trigger errors.

Input units can influence how subsequent input units are processed. Filtered iterators may therefore not be appropriate for applications that must successfully process all input units. For example, an application that first updates, then queries, a database may need to successfully process all of the updates so that the subsequent queries execute properly. Filtered iterators may also not be appropriate for applications that should terminate when they encounter an unanticipated error. A compiler, for example, should usually terminate when it encounters an internal compiler error rather than silently generate potentially incorrect code. In general, we expect developers to match the needs of their applications against the capabilities of filtered iterators to determine whether filtered iterators are appropriate for their application.

Empirical Evaluation: We evaluate the effectiveness of filtered iterators in a controlled experiment. The experiment uses paired human developers to test hypotheses regarding the ability of filtered iterators to deliver smaller, more robust programs with fewer defects. 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 detected statistically significant differences between the two groups in terms of program defects, program survival, and program complexity (see Section 4). These results support the hypothesis that filtered iterators can enable a range of developers to produce more robust programs with significantly fewer defects than comparable developers using standard language constructs.

[1] Note that *bad* input units may differ from *illegal* (malformed) input units that violate the input format specification. See Section 4 for definitions.

[2] To help developers debug programs, the language runtime generates a log that contains all errors that occurred during execution. See Section 3.3.

1.2 Contributions

This paper makes the following contributions:

Filtered Iterators: We present filtered iterators, a novel control structure as the synergy between atomicity, exception handling, and input processing.

Empirical Evaluation: We present results from an empirical evaluation of filtered iterators. The statistically significant results highlight the effectiveness of filtered iterators in eliminating many of the difficulties that developers face when developing input processing code using standard constructs.

We also briefly discuss our experience using filtered iterators to develop programs for a variety of input formats including the PCAP, PNG, ZIP, JSON, and OBJ formats.

2 Example

We illustrate filtered iterators by contrasting two implementations of a bitmap thumbnail generator program. The first implementation uses conventional C language constructs to handle errors *explicitly* (Conventional C Version). The second implementation uses filtered iterators to handle errors *implicitly* (C/RIFL Version).

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 1a 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: (1) A string image name, which contains no space or newline characters and is 1–10 characters long. (2) An integer scaling factor, s. (3) An integer height, h. (4) An integer width, w. (5) $h \cdot w$ consecutive digits, ranging from '0'…'9', that each represents the color of a pixel in the original image. The pixels represent the image by forming a matrix of h rows and w columns. Pixels are ordered as follows: inside each row, pixels are ordered from left to right; the rows are grouped together and ordered from top to bottom.

Program Output: Figure 1b presents the example output. Each line describes a thumbnail image with the following components, separated by a single space character: (1) The image name. (2) $\lfloor h/s \rfloor \lfloor w/s \rfloor$ digits that each represents the value of a pixel in the thumbnail.

Robustness: Like other image file formats [2, 11, 21], the input files in our example may contain multiple images. To maximize utility, the program should continue to generate thumbnails for other images even after encountering one for which (for whatever reason) it cannot generate a thumbnail.

The example input in Figure 1a contains illegal lines that violate 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. Some other lines in the example input are legal but rare. They satisfy the specification but may still trigger errors in defective implementations. 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 1b excludes thumbnails for these kinds of illegal or rare images.

```
1  Img1 2 2 2 1234
2  Img2 2 4 4 1234567890123456
3  CharS b 2 2 1234
4  CharPix 2 2 2 12a4
5  BufOvfVeryLongName 2 2 2 1234
6  2 2 2 1234
7  Div0S 0 2 2 1234
8  Div0H 2 0 2
9  HeapOvf 2 60000 1000 1234
10 BufOvfInt 2 16 268435457
       12345678901234567890
11 Img3 2 1 2 12
12 Img4 3 3 4 123456789012
```

(a) Example input

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

(b) Example output

Figure 1. Example input and output for thumbnail programs

2.1 Conventional C Version

Figure 2 presents a conventional C 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: **Vital** code that prevents crashes is in red. **Integrity** code that prevents input misinterpretation and input desynchronization is in brown. **Service** code that cleans up resources is in purple.

Table 1 enumerates the error-handling code in this implementation. Each row describes a piece of code. The columns are as follows: the identifier number (Id), location (Lines), importance category (Category), and functionality (Purpose).

2.2 C/RIFL Version

Figure 3 presents another version of the thumbnail program (with the same functionality) using the `inspectt` construct as a filtered iterator. The program uses C syntax augmented with the RIFL `inspectt` loop construct and associated semantics. These semantics include (1) atomic execution of the loop body, (2) array bounds and assertion checks, and (3) automatic error recovery when an error or assertion failure is detected in the loop body. [3]

[3] The rest of this paper evaluates filtered iterators using the RIFL language, which is presented in Section 3. A RIFL version of the thumbnail program is available in the RIFL technical report [42].

```
1  FILE *f;
2  void flush (char x) {                                  // I1
3    while (x!=EOF && x!='\n') { x=fgetc(f); }            // I1
4  }
5  int rdint () {
6    int x=0; char dgt=fgetc(f);
7    while (dgt!=' ') {
8      if (dgt == EOF || dgt<'0' || dgt>'9') {            // I2
9        fseek(f,-1, SEEK_CUR);                           // I3
10       return -1;                                       // I4
11     }
12     x=x*10+(dgt-'0'); dgt=fgetc(f);
13   }
14   return x;
15 }
16 int main (){
17   f = fopen("images.txt", "r");
18   if (!f) { return -1; }                                // V1
19   char name[10];
20   char x;
21   while ((x=fgetc(f)) != EOF) {
22     int bad=0; int i=0;                                 // I5
23     while (x!=' ') {
24       if (x==EOF || x=='\n' || i>=10) {                 // V2,I6,I7
25         bad=1; break;                                   // I8
26       }
27       name[i]=x; i=i+1; x=fgetc(f);
28     }
29     if (i<=0 || bad) { flush(x); continue; }            // I9
30     while (i<10) { name[i]=0; i=i+1; }
31     int s=rdint(); int h=rdint(); int w=rdint();
32     if (s<=0||h<=0||w<=0||(h*w)/h!=w) {                 // V3,V4,I10
33       flush(0); continue;                               // I11
34     }
35     char *pix=(char *)malloc(h*w);
36     if (!pix) { flush(0); continue; }                   // V5
37     i=0;
38     while (i<h && !bad) {                               // I12
39       int j=0;
40       while (j<w) {
41         x=fgetc(f);
42         if (x==EOF||x<'0'||x>'9'){bad=1;break;}         // I13
43         pix[i*w+j]=x; j=j+1;
44       }
45       i=i+1;
46     }
47     flush(x);                                           // I14
48     if (bad) { free(pix); continue; }                   // I15,S1
49     printf("%s ", name);
50     // << compute and print averages, code omitted >>
51     putchar('\n'); free(pix);
52   }
53   fclose(f);
54   return 0;
55 }
```

Figure 2. A conventional C thumbnail implementation

In particular, the (simplified) structure of an `inspectt` loop is as follows:

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

This loop iterates through file `f`, executing a block of code for each input unit. A delimiter value du defines the boundaries between input units. In each iteration, `inspectt` nullifies the effects of all detectable errors by processing each input unit *atomically* so that all updates from each input unit either commit or abort. After each iteration, `inspectt` always advances the input file pointer and positions it after

```
1  FILE *f;
2  int rdint () {
3    int x=0; char dgt=fgetc(f);
4    while (dgt!=' ') {
5      assert(dgt>='0' && dgt<='9');
6      x=x*10+(dgt-'0'); dgt=fgetc(f);
7    }
8    return x;
9  }
10 int main () {
11   f = fopen("images.txt", "r");
12   char name[10];
13   char x;
14   inspectt (f,'\n') {
15     int i=0; x=fgetc(f);
16     while (x!=' ') {
17       name[i]=x; i=i+1; x=fgetc(f);
18     }
19     assert(i>=1);
20     while (i<10) { name[i]=0; i=i+1; }
21     int s=rdint(); int h=rdint(); int w=rdint();
22     char *pix=(char *)malloc(h*w);
23     i=0;
24     while (i<h) {
25       int j=0;
26       while (j<w) {
27         x=fgetc(f);
28         assert(x>='0' && x<='9');
29         pix[i*w+j]=x; j=j+1;
30       }
31       i=i+1;
32     }
33     printf("%s ", name);
34     // << compute and print averages, code omitted >>
35     putchar('\n'); free(pix);
36   }
37   fclose(f);
38   return 0;
39 }
```

Figure 3. A C/RIFL thumbnail implementation

the delimiter. This mechanism ensures that each iteration starts reading input at the beginning of the next input unit, which prevents the C/RIFL program from desynchronization, which occurs when a program fails to completely read in a bad input unit, leaving the file pointer incorrectly positioned in the middle of the preceding bad input unit (instead of correctly positioned at the start of the next input unit). This semantics skips bad input units (which trigger errors or assertion failures) so that the program executes as if these bad input units were not present in the input at all.

Eliminated Checks: We next discuss checks and statements that are required in the conventional C version but are unnecessary in the C/RIFL version. Checks V1–V5 (see Table 1), which are vital in the conventional C version, are unnecessary in the C/RIFL version because filtered iterators at runtime implicitly trigger an error on invalid file accesses, failed heap allocations, out-of-bounds array accesses, and divisions by zero. The `inspectt` loop starting from line 14 implicitly handles all these errors by discarding bad input units atomically. On the other hand, if the conventional C version omits any of the vital error-handling code, the program would crash from bad inputs:

Table 1. Error-handling code in the conventional C thumbnail program.

Id	Lines	Category	Purpose
V1	18	vital	avoid invalid file accesses
V2	24 (`i>=10`)	vital	avoid out-of-bounds array accesses
V3	32 (`s<=0, h<=0`)	vital	avoid divisions by zero
V4	32 (`((h*w)/h!=w`)	vital	avoid integer overflows that can cause out-of-bounds array accesses
V5	36	vital	avoid null array accesses
I1	2–4	integrity	flush trailing bytes for current unit (helper function)
I2	8 (`dgt==EOF`)	integrity	identify incomplete units at the ends of files
I3	9	integrity	avoid mixing bad units with following units
I4	10	integrity	identify bad units
I5	22 (`bad=0;`)	integrity	default to good units
I6	24 (`x==EOF`)	integrity	identify incomplete units at the ends of files
I7	24 (`x=='\n'`)	integrity	avoid mixing bad units with following units
I8	25	integrity	skip bad units
I9	29	integrity	skip bad units
I10	32 (`w<=0`)	integrity	identify bad units
I11	33	integrity	skip bad units
I12	38 (`!bad`)	integrity	avoid mixing bad units with following units
I13	42	integrity	identify bad units
I14	47	integrity	avoid mixing bad units with following units
I15	48 (`bad`)	integrity	skip bad units
S1	48 (`free`)	service	avoid memory leak on bad units

- **Direct out-of-bounds array access:** Without check V2, the conventional C version writes beyond array `name` on line 27 when an image name is longer than the array size.
- **Indirect out-of-bounds array access caused by integer overflow:** Without check V4, the conventional C 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 V3, the conventional C version triggers division-by-zero errors on line 32 or in the code for line 50 when an image contains a zero height or a zero scaling factor.
- **Null array access:** Without check V5, the conventional C version writes to a null array on line 43 when the memory allocation fails on line 35.
- **Invalid file access:** Without check V1, the conventional C version reads from a null file pointer on line 21 when the opening operation fails on line 17.

Apart from freeing the developers from having to write otherwise vital error-handling code above, filtered iterators also reduce the need for other code in the conventional C version. With automatic error handling, there is no need to explicitly maintain the integrity of input units as in (1) checks I2, I6, I7, I9, I10, I12, and I15, or (2) statements I1, I3–I5, I8, I11, I13, and I14. Statement S1 is also unnecessary in the C/RIFL version because the `inspectt` loop discards bad input units atomically, preventing a memory leak in this situation. On the other hand, the conventional C version would be defective without these checks and statements:

- **Input misinterpretation and desynchronization:** The code I2, I4–I6, I9, I10, and I15 detect bad input units. Code I1, I3, I7, I8, and I11–I14 isolate bad units from other inputs.
- **Memory leak:** Without statement S1, the conventional C version leaks memory on images with corrupted pixel values. Memory leaks degrade the quality of service and may result in allocation failures.

The C/RIFL version has simpler control flow and fewer statements than the conventional C version.

$$
\begin{aligned}
\text{Prog} &:= \text{Stmt} \mid \texttt{func}\ q(x)\{\text{Stmt};\texttt{return}\ y\};\text{Prog}\\
\text{Exp} &:= n \mid x \mid \text{Exp op Exp} \mid a[\text{Exp}] \mid \texttt{valid}(a) \mid \texttt{end}(f) \mid \texttt{pos}(f)\\
\text{Stmt} &:= x = \text{Exp} \mid a = \texttt{malloc}(\text{Exp}) \mid \texttt{free}(a) \mid a[\text{Exp}] = \text{Exp}\\
&\quad \mid x = q(\text{Exp}) \mid \texttt{seek}(f, \text{Exp}) \mid x = \texttt{read}(f) \mid \texttt{assert}(\text{Exp})\\
&\quad \mid \text{Stmt}; \text{Stmt} \mid \texttt{if}(\text{Exp})\{\text{Stmt}\}\texttt{else}\{\text{Stmt}\} \mid \texttt{while}(\text{Exp})\{\text{Stmt}\}\\
&\quad \mid \texttt{inspectt}(\text{Exp}, f, d_u, d_s)\{\text{Stmt}\} \mid f = \texttt{opent}(str)\\
&\quad \mid \texttt{inspectb}(\text{Exp}, f, \text{Exp}, \text{Exp}, \text{Exp})\{\text{Stmt}\} \mid f = \texttt{openb}(str)
\end{aligned}
$$

$$
x, y \in \mathit{IVar} \qquad a, d_u, d_s \in \mathit{AVar} \qquad f \in \mathit{FVar}
$$
$$
q \in \text{function names} \qquad n \in \mathit{Int} \qquad \mathit{str} \in \mathit{String}
$$

Figure 4. Abstract syntax

3 The RIFL Language

To support our experimental evaluation of filtered iterators, we design the Robust Input Filtering Language (RIFL). RIFL is a simple imperative language inspired by C. RIFL has two filtered iterator constructs: the `inspectt` construct for text files and the `inspectb` construct for binary files.

3.1 Syntax

Figure 4 presents the syntax of RIFL. The structure of inspect loops for text inputs is

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

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

The structure of inspect loops for binary inputs is

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

which similarly iterates through input units in a binary file `f`. The contents of each input unit is computed from a length field in `f`, which is specified by the offset parameter `o` and the width parameter `w`, and the extra length parameter `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`.

To allow more sophisticated ways to specify the boundaries of input units, it is possible to incorporate I/O libraries that decompose inputs into input units.

3.2 Semantics

We next discuss the RIFL semantics. The full operational semantics is presented in the RIFL technical report [42].

Atomic Execution and Filtering: Inspect loops handle errors by filtering out bad input units. The RIFL runtime includes checks for execution integrity violations such as out-of-bounds accesses, null pointer dereferences, divisions by zero, and memory allocation failures. Of course, not all application errors manifest as these errors. RIFL therefore also supports *assertions*, which enable developers to explicitly specify application-level validation checks in inspect loops.

When an inspect loop iteration triggers an error or assertion failure, the RIFL runtime handles the error or failure automatically by (1) identifying the bad input unit, (2) advancing the file pointer past the bad input unit, (3) restoring the rest of the program state, such as externally visible memory, by rolling back to the old state when entering the current iteration, (4) discarding any externally visible outputs that the current iteration produced, and (5) restarting the program execution. Conceptually, the program restarts the original inspect loop iteration, but with the file pointer changed to the start of the next input unit (unless the bad input unit is the last input unit in the file).

Atomicity is known to be difficult for developers to implement correctly [10, 13]. Filtered iterators eliminate these difficulties because the atomicity is implemented systematically by the RIFL implementation.

Nesting and Recursion: Nested inspect loops can process hierarchical input structures. Nested filtered iterators are supported with nested transactions. For example, the JavaScript Object Notation (JSON) [12] input format contains nested objects. The objects are delimited by brace characters. A RIFL program that processes JSON input may repeatedly use the right brace character as the delimiter for objects (input units) across all nesting layers. The program may contain a recursive function where an inspect loop recursively parses the nested inner input units.

3.3 Implementation

Besides the RIFL operational semantics, we developed two RIFL implementations: (1) a full RIFL interpreter in OCaml and (2) a compiler in Scala that generates C code for RIFL programs with `inspectt` loops. The generated C programs (1) capture runtime errors with dynamic checks and (2) implement atomicity by explicitly saving state and invoking the built-in `setjmp()` and `longjmp()` functions. While RIFL can use any atomicity mechanism [20, 41], both our current implementations use standard checkpointing techniques to restore externally visible data structures if an iteration encounters a bad input unit and aborts. Both implementations use standard logging techniques to delay externally visible outputs until each iteration commits [20].

While performance is not the focus of our current implementations, we did measure the performance of the two C programs compiled from the two RIFL programs in Section 2. For a range of input sizes, the RIFL version compiled program's execution time was 34%–51% longer than the conventional version compiled program's execution time. Most of the overhead was generated by the RIFL runtime checks, occurring mostly in the main computation after parsing the input (line 32 in Figure 3). This overhead is consistent with prior research on the overhead of memory safety checks [31, 33, 39] and could be reduced by applying prior techniques developed for optimizing these checks. The overhead caused by enforcing atomicity was small, because each inspect loop iteration has little externally visible data to back up and restore. Overall, we believe this overhead is reasonable as a trade off for the increased robustness.

By design, filtered iterators enable programs to survive errors and continue execution. To aid debugging and correcting these errors, the filtered iterator runtime generates an exhaustive error log that contains all errors that occurred during execution. Developers can examine the error log and take appropriate action if desired. The runtime can also be configured to terminate execution immediately on any error. This configuration may be especially helpful during software development (as opposed to production).

4 Controlled Experiment

We evaluate the design of filtered iterators with a controlled experiment that uses pairwise comparison [46, 54] to evaluate its use in program development. We are interested in the following research questions:

- **RQ1:** Does `inspect` reduce program *defects*?
- **RQ2:** Does `inspect` increase program *survival*?
- **RQ3:** Does `inspect` reduce *wrong outputs*?
- **RQ4:** Does `inspect` reduce *wrong outputs* even if we apply failure-oblivious computing [38] to the control group?
- **RQ5:** Does `inspect` reduce program *complexity*?

To address these questions, we define input categories based on their accordance to the input format specification: **Common legal** inputs satisfy the input specification and belong to common scenarios that are typically correctly handled by the main functionality of programs. **Rare legal** inputs satisfy the input specification while containing corner

cases that may be incorrectly handled by developers. **Illegal** (malformed) inputs violate the input specification.

Besides these categories, **bad** inputs are inputs that trigger errors or assertion failures in program execution, whether they are illegal or not. Filtered iterators discard bad input units. Filtered iterators are not specifically designed to find and discard all illegal inputs or input units, although in many cases they may do so. They are instead designed to find and discard input units, illegal or not, that the program cannot process successfully.

4.1 Methodology

Experimental Design: We defined an input processing programming task, specifically, a thumbnail generator for files that may contain multiple images, as in Section 2. We recruited subjects to write the program. These subjects were drawn from the population of active developers at MIT, including doctoral students and post-doctoral researchers. The study was approved by the institutional review board.

The Wilcoxon signed-rank test [53] tests the location shift of paired samples. As directed by the test, we first manually matched pairs of demographically similar subjects based on educational background, programming experience in the past five years, and knowledge of C/C++.[4] We next randomly assigned each pair of subjects to either the inspect group or the control group. Both groups performed the programming task on a virtual machine. We then analyzed the resulting developer programs with the Wilcoxon signed-rank test.

Independent Variable: The only difference between the two groups was the use of the `inspect` construct. The inspect group used the full RIFL language. The control group used RIFL without the `inspect` construct, but with standard constructs for control flow and error handling. To minimize other differences, we purposefully disabled the special error logs for `inspect` (which, if enabled, would provide the inspect group with additional helpful information).

Dependent Variables: Before performing experiments, we decided to analyze all programs focusing on their defects, behavior on a range of inputs, and program complexity. We were 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 column presents abbreviations and the second column describes the defects in detail. We say that a program has a defect if the defect appears anywhere in the program.
- **Behavior on test inputs:** To address **RQ2** and **RQ3**, we predefined 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 inputs include *common legal* inputs that test the main functionality, *rare legal* inputs that test corner cases that still satisfy the input specification but may be incorrectly handled by developers, and *illegal* inputs that test the error-detection and error-handling code. We extended the list of inputs to trigger new defects observed from code review. All of these inputs and their correct outputs are available [42].
- **Comparison with failure-oblivious executions:** To address **RQ4**, we implemented a failure-oblivious computing (FOC) [38] version for the control group's language. This implementation has the same behavior as the original control group language in benign situations and differs in erroneous situations that would otherwise cause crashes. Specifically, the FOC implementation (1) returns value zero for divide-by-zero expressions, out-of-bounds array reads, and null array reads and (2) silently ignores out-of-bounds array writes and null array writes. For each test input, we observed the behavior of the control group programs running with failure-oblivious computing. For reference, we also analyzed the behavior of programs that entered

Table 2. Possible fatal defects. These defects are predefined, except for the underlined defect which is new observed.

Defect	Description
AWL	Out-of-bounds array write when reading input, triggered by input fields that are longer than an input buffer.
AWO	Out-of-bounds array write when reading input, triggered by an integer overflow that causes overly small memory allocation.
ARL	Out-of-bounds array read during computation, triggered by image dimensions that are too large for an input buffer.
ARO	Out-of-bounds 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. These defects are predefined, except for the underlined defects which are new observed.

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 desynchronization 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.

[4] To minimize noise, we matched C/C++ knowledge because RIFL resembles C/C++ in syntax, program layout, and memory management interface.

infinite loops, assuming the presence of tools [14] that help programs (1) escape from infinite loops and (2) continue executing the instructions that follow the escaped loop.

- **Program complexity:** To address **RQ5**, we measured program complexity in terms of predefined metrics. **Control flow** metrics include (1) cyclomatic complexity [28], which indicates the complexity of the decision structure of a program, and (2) number of conditional clauses in conditional predicates and assertion statements. **Code size** metrics include (1) number of lines in the source code, (2) number of all statements, and (3) number of unconditional 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 example program, and writes two small programs that may be helpful for implementing the thumbnail generator. These two tasks are designed to interactively *teach* subjects the experimental languages and to *calibrate* the two groups in terms of language understanding before they start the task.

The specification for the thumbnail generator task explains the program functionality with typical inputs and explicitly states that the program should be able to handle arbitrary inputs by skipping malformed images. Complete instructions for the experiment are available [5, 42]. The experiments are unlimited in duration. To gain more insight into the developers' experience, we recorded and examined full screen recordings of the developers as they worked.

Prior to collecting data, we used a pilot experiment to estimate the effect size and the number of subjects needed. We decided to recruit ten subjects.

4.2 Results

Subjects: Ten 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 denote each subject with a letter that stands for the group followed by a number that stands for the pair. Letters "i" and "c" denote the inspect group and the control group, respectively. The subjects were matched up into five pairs (numbered 1 through 5), based on the similarity of their backgrounds. Among the five pairs, four pairs (2; 3; 4; 5) received bachelor's degrees in Computer Science or equivalent majors and one pair (1) in Electrical Engineering. Two pairs (1; 2) received C/C++ education in the past five years and primarily write scientific-computing programs in Python and Matlab. The other three pairs (3; 4; 5) have been proficient with C/C++ for at least five years. Two of them (3; 4) primarily write system-level programs in C/C++ and Go. The other pair (5) has trained for competitive programming in C/C++.

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 (i4; c5) 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; c1; c2; i3) were fixed during our analysis.

Table 4. Defects in inspect (i1–i5) and control (c1–c5) group programs. Letter "X" denotes the existence of a defect.

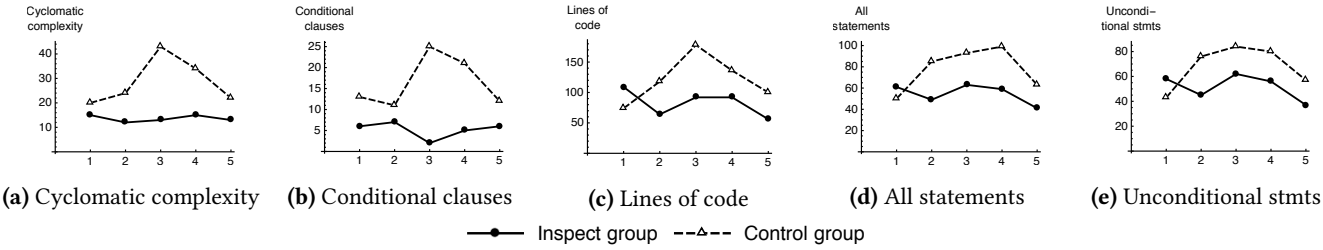
Defect	i1	i2	i3	i4	i5	c1	c2	c3	c4	c5
AWL						X	X		X	
AWO										
ARL						X				
ARO							X	X	X	X
DS						X	X		X	X
DD										
NA						X	X	X	X	X
IL							X			
Fatal	**0**	**0**	**0**	**0**	**0**	**4**	**5**	**2**	**4**	**3**
MP				X	X	X			X	X
MS	X						X		X	
WP							X			X
WS							X			X
WM			X	X	X	X	X		X	X
WA									X	X
Other	**1**	**0**	**1**	**2**	**2**	**2**	**4**	**0**	**4**	**5**
Total	**1**	**0**	**1**	**2**	**2**	**6**	**9**	**2**	**8**	**8**

Most subjects took comparable time, between one and two hours, to finish the entire experiment including tutorial. Two subjects (c1; c2) took between two and three hours.

Measurements: We provide the full raw data online [5]. Table 4 presents the defects found in all developer programs. Each column represents a program by the name of the developer. The rows are: the list of fatal defects (first eight rows), number of fatal defects ("Fatal"), list of other defects (next six rows), number of other defects ("Other"), and number of all defects ("Total"). Table 5 presents the behavior of these programs processing the test inputs. Each column represents a program. We denote the failure-oblivious executions of the control group programs with "foc" followed by the pair number. The rows are: behavior on common legal inputs (the first row), behavior on rare legal inputs (next four rows), number of crashes or infinite loops and the number of incorrect outputs on rare legal inputs ("Crash/Loop rare" and "Incorrect/Explode rare"), behavior on illegal inputs (next 21 rows), number of crashes or infinite loops and number of incorrect outputs on illegal inputs ("Crash/Loop illegal" and "Incorrect/Explode illegal"), and number of crashes or infinite loops and the number of incorrect outputs on all inputs ("Crash/Loop total" and "Incorrect/Explode total"). We define an incorrect output to *explode* if the output size is proportional to an input value, instead of to the input size. The reasons for the incorrect behavior in Table 5 are the defects in Table 4. Figure 5 presents the complexity measurements for all developer programs. Vertical axes in Figures 5a, 5b, 5c, 5d, and 5e represent cyclomatic complexity, number of

Table 5. Behavior of the inspect group programs (i1–i5), original control group programs (c1–c5), and failure-obliviously executed control group programs (foc1–foc5) processing test inputs. These inputs are predefined, except for the underlined input which is new developed. Letters "C", "L", "W", and "E" denote crashing, entering an infinite loop, producing incorrect (wrong) and small outputs, and producing incorrect and exploding outputs, respectively. Letter "s" denotes data desynchronization for at least one subsequent input unit. Letter "a" denotes aborting. Combination "Ls" denotes that the program enters an infinite loop and that the program would desynchronize after escaping from the infinite loop. Empty cells denote correct behavior.

Input	Category	i1	i2	i3	i4	i5	c1	c2	c3	c4	c5	foc1	foc2	foc3	foc4	foc5
good	common legal															
bufovfverylongname	rare legal							C		C			W			
div0h	rare legal															
div0w	rare legal															
heapovf2	rare legal						C	C	C	C	C	W	W	W		W
Crash/Loop rare legal		**0**	**0**	**0**	**0**	**0**	**1**	**2**	**1**	**2**	**1**	**0**	**0**	**0**	**0**	**0**
Incorrect/Explode rare legal		**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**2**	**1**	**0**	**1**
bufovfint1	illegal						C	C		C	C	E	E		E	Es
bufovfint2	illegal						C	C		C	C	E	E		E	Es
bufovfint3	illegal						C	C	C	C	C	E		E	E	E
bufovfint4	illegal						C	C		C	C	E	W		E	Es
charh	illegal					W	W	Ws		W		W	Ws		W	
charpix1	illegal			W	W	W	W	Ws			Wa	W	Ws			Wa
charpix2	illegal							Ls			Wa		Ls			Wa
chars	illegal			W			W	C			W	W	W			W
chartrail	illegal							Ws			Ws		Ws			Ws
charw1	illegal					W	W	Ws		W		W	Ws		W	
charw2	illegal						C	W			Ws	W	W			Ws
div0s	illegal						C	C		C	C	W	W		W	W
empty	illegal															
heapovf1	illegal						C	C	C	C	C	E	Ls			E
intovf	illegal															
long	illegal						C	C		C	Ws		Ws			Ws
nullint	illegal					W	W	Ws		Wa	Ws	W	Ws		Wa	Ws
short1	illegal						W	Ls			W	W	Ls			W
short2	illegal						C	Ws			Ws	W	Ws			Ws
short3	illegal						W	Ls			W	W	Ls			W
short4	illegal						C	C			Ws	W	Ws			Ws
Crash/Loop illegal		**0**	**0**	**0**	**0**	**0**	**10**	**12**	**2**	**7**	**6**	**0**	**4**	**0**	**0**	**0**
Incorrect/Explode illegal		**0**	**0**	**2**	**1**	**4**	**7**	**7**	**0**	**3**	**11**	**16**	**14**	**1**	**8**	**17**
Crash/Loop total		**0**	**0**	**0**	**0**	**0**	**11**	**14**	**3**	**9**	**7**	**0**	**4**	**0**	**0**	**0**
Incorrect/Explode total		**0**	**0**	**2**	**1**	**4**	**7**	**7**	**0**	**3**	**11**	**17**	**16**	**2**	**8**	**18**



(a) Cyclomatic complexity **(b)** Conditional clauses **(c)** Lines of code **(d)** All statements **(e)** Unconditional stmts

Figure 5. Complexity measurements for developer study

conditional clauses, lines of code, number of all statements, and number of unconditional statements, respectively. Horizontal axes represent the five pairs of subjects.

Statistical Analysis: We applied the one-sided Wilcoxon signed-rank tests [53] on all measurements using R [36]. We assume that the differences are comparable across pairs [46]. According to the test results, we observed statistical significance to reject null hypotheses where the p-values are less than the standard significance level, 0.05. Specifically:

- **Defects, RQ1:** The inspect group has significantly fewer defects ($p = 0.029$) and fewer fatal defects ($p = 0.029$) than the control group. See Table 4.

- **Survival, RQ2:** The inspect group crashes or enters infinite loops on significantly fewer occasions than the control group when processing rare legal inputs ($p = 0.027$), illegal inputs ($p = 0.031$), and all inputs ($p = 0.031$). See Table 5.
- **Incorrect outputs with FOC, RQ4:** The inspect group produces fewer incorrect outputs than the failure-oblivious executions of the control group programs when processing rare inputs ($p = 0.044$). See Table 5.
- **Program complexity, RQ5:** The inspect group programs have significantly smaller cyclomatic complexity ($p = 0.031$) and fewer conditional clauses ($p = 0.031$) than the control group. See Figures 5a and 5b.

Apart from these statistically significant results, we also observed the following interesting tendencies:

- **Defects, RQ1:** The inspect group has fewer non-fatal defects than the control group ($p = 0.068$). See Table 4.
- **Incorrect outputs, RQ3:** The inspect group produces fewer incorrect outputs than the original control group when processing illegal inputs ($p = 0.064$). See Table 5.
- **Incorrect outputs with FOC, RQ4:** The inspect group produces fewer incorrect outputs than the failure-oblivious control group when processing illegal inputs ($p = 0.063$) and all inputs ($p = 0.050$). See Table 5.
- **Program complexity, RQ5:** The inspect group programs have fewer lines of code ($p = 0.052$), fewer all statements ($p = 0.063$), and fewer unconditional statements ($p = 0.063$). See Figures 5c, 5d, and 5e.

Our overall conclusion is that these results support the hypothesis that filtered iterators can enable a range of developers to produce 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 security vulnerabilities, present in the control group programs are consistent with defects observed in input processing code more broadly.

4.3 Discussion

Program Robustness: Defects, survival, and incorrect outputs are three measures of program robustness. The experimental results show that, for our programs, filtered iterators reduced defects, increased survival, and reduced incorrect outputs. We attribute these improvements to the ability of filtered iterators to (1) eliminate program crashes completely, (2) discard bad input units atomically, which enables more predictable program behavior by preventing partially corrupted program state or outputs, (3) advance to the next input unit automatically, which prevents desynchronization on errors, and (4) reduce the program complexity, which allows developers to reason about their programs more easily and to produce fewer non-fatal defects.

Program Complexity: The results show that filtered iterators significantly reduced the cyclomatic complexity and the number of conditional clauses. The inspect group programs contain more straightforward code, using *assertions* to replace many of the more convoluted error-handling checks that the control group programs need. Also note that, for cyclomatic complexity, the inspect group cluster in a smaller range ($\sigma = 1.34$) while the control group scatter in a larger range ($\sigma = 9.69$). This difference in aggregation indicates that filtered iterators reduced both the complexity and the number of different designs for decision structures. As an insight, with the language automatically handling exceptional situations, developers can be relieved from anticipating certain corner cases and focus more on the main functionality.

The code size—lines of code, all statements, and unconditional statements—was also reduced, though not statistically significant. We attribute these improvements to the ability of filtered iterators to (1) ensure atomic recovery, which prevents bad input units from corrupting the data structures accessed across multiple input units even when the inspect group programs omit the code for maintaining these data structures, and (2) eliminate the code that identifies boundaries between input units and prevents desynchronization in case of bad input units. We attribute the lack of significance here to the noise caused by various approaches to implementing the same functionality. For example, program i1 is longer than program c1 because (1) i1 elaborates intermediate computations as individual statements on temporary data structures and (2) c1 reads all fields in a line using a loop, which reuses code. Despite these implementation variations, the inspect group still tends to have shorter code.

Defects: The control group programs contain a substantial number of input processing defects, highlighting the difficulty of developing robust input processing code using current language constructs.

Threats to Validity: Using standalone programming tasks could limit the ability to generalize the results to industrial practice. Participant variation might affect results. We mitigated by manually matching similar participants and randomly assigning each pair into groups [53]. One group might learn faster. We mitigated by having participants finish tutorial tasks before starting the thumbnail task. A new language could affect one group more. We mitigated by matching participants into pairs with similar language experience and by manually examining screen recordings to verify that participants worked with the new language without changing their programs' behavioral framework. Participants might guess our hypotheses and be more or less careful handling errors than normal. We mitigated by not revealing our goals and design to either group. We provide full design and raw data [5] for reproduction.

5 Other Input File Formats

In addition to the controlled experiment, we also investigated the use of RIFL for the following input file formats. Binary formats include network traffic capture file format PCAP [6, 16], image format PNG [11], and file compression

format ZIP [1]. Text formats include JavaScript object format JSON [12], three-dimensional object geometry format OBJ [3], and table format CSV [40]. Each format is processed by a benchmark program. An analysis shows that, compared to the conventional versions, the full RIFL versions have on average 62.2% smaller cyclomatic complexity, 59.1% fewer conditional clauses, 41.9% fewer lines of code, 38.5% fewer all statements, and 34.2% fewer unconditional statements. A more exhaustive evaluation of the data from these experiments is available in our technical report [42].

6 Related Work

Language Constructs: Iterators [25] generalize loops over collections. Filter functions [48] apply a predicate to extract a subset of elements from a collection. In contrast to explicit filtering predicates, filtered iterators use the program execution as an implicit filter. The result is automatic detection and recovery from a range of errors. Exception handling [19] can improve program structure by separating common-case and error-handling code. In contrast, filtered iterators additionally (1) decompose inputs into input units, (2) provide transactional semantics that atomically discards bad input units, and (3) provide for continued processing of subsequent input units. Implementing filtered iterator functionality using exceptions would require changing the language semantics to support (1) atomic execution, that is, discarding bad input units atomically and continuing execution, and (2) the decomposition of inputs into input units. Recovery blocks [7] and N-version programming [8] tolerate software faults using multiple implementations of the same component. Bristlecone [17] ensures error-free execution using decoupled transactional tasks according to high-level task specifications. Shinnar et al. [43] propose the `try_all` construct for C# programs. Warth et al. [51] propose the "worlds" construct for JavaScript programs. These constructs combine exception handling with rollback of program state, with the goal of ensuring data structure consistency and controlling the scope of in-memory side effects in the presence of exceptions. The motivation for filtered iterators is different. Specifically, filtered iterators enable robust execution for programs that process input units. To this end, filtered iterators, `try_all`, and "worlds" all preserve data structure consistency and control side effects. But filtered iterators also detect and discard bad input units, roll back outputs, and ensure that every iteration begins its execution at the start of the next input unit. We also evaluate the effectiveness of filtered iterators using a controlled experiment.

Recovery by Manipulating Execution: Failure-oblivious techniques [27, 38] purposefully change the program semantics to survive inputs that the program would otherwise be unable to process. Error virtualization [44] 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 uncertain impacts on the semantics of the program. Filtered iterators integrate error handling in the language semantics.

Recovery with Replay: Transactional recovery techniques such as backward recovery [15] and forward recovery [23] recover programs from transient errors. Rx [35] rolls back and replays programs in new configurations to survive failures. These techniques rely on the existence of alternative non-deterministic computations.

Empirical Studies That Compare Languages: Empirical studies have compared programming languages by analyzing code repositories [9, 32, 37] or performing controlled experiments on developers [22, 34, 49]. We similarly use a rigorous controlled experiment on developers and detect statistical significance. One difference is that our experiment evaluates the design of a novel language construct, while the cited experiments evaluate existing languages.

7 Conclusion

Input processing defects are a common source of software errors and security vulnerabilities. We propose a new language construct, filtered iterators, to help developing reliable and robust programs. The statistically significant results from a developer study demonstrate the empirical benefits that filtered iterators can provide.

Acknowledgments

We thank Deokhwan Kim, Julia Rubin, and the anonymous reviewers for their insightful comments. This research was supported by DARPA (Grant FA8750-14-2-0242).

References

[1] 1989. .ZIP Application Note. https://www.pkware.com/support/zip-app-note/. (1989).
[2] 1990. GRAPHICS INTERCHANGE FORMAT Version 89a. http://www.w3.org/Graphics/GIF/spec-gif89a.txt. (1990).
[3] 1996. Wavefront OBJ File Format Summary. http://www.fileformat.info/format/wavefrontobj/egff.htm. (1996).
[4] 2017. CVE – Common Vulnerabilities and Exposures. http://cve.mitre.org/. (2017).
[5] 2017. SLE 2017 RIFL Artifact. http://people.csail.mit.edu/jiasi/sle2017_rifl_artifact/ and https://people.csail.mit.edu/rinard/paper/sle17.rifl.artifact. (2017).
[6] 2017. Wireshark. https://www.wireshark.org/. (2017).
[7] T. Anderson and R. Kerr. 1976. Recovery Blocks in Action: A System Supporting High Reliability. In *Proceedings of the 2Nd International Conference on Software Engineering (ICSE '76)*. 447–457.
[8] A. Avizienis. 1985. The N-Version Approach to Fault-Tolerant Software. *IEEE Trans. Softw. Eng.* 11, 12 (Dec. 1985), 1491–1501.
[9] P. Bhattacharya and I. Neamtiu. 2011. Assessing Programming Language Impact on Development and Maintenance: A Study on C and C++. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. 171–180.
[10] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. 2014. DoubleChecker: Efficient Sound and Precise Atomicity Checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. 28–39.
[11] T. Boutell. 1997. PNG (Portable Network Graphics) Specification Version 1.0. (1997).

[12] T Bray. 2014. JavaScript Object Notation (JSON) Data Interchange Format. http://www.rfc-editor.org/rfc/rfc7159.txt. (March 2014). RFC 7159.

[13] J. Burnim, G. Necula, and K. Sen. 2011. Specifying and Checking Semantic Atomicity for Multithreaded Programs. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 79–90.

[14] M. Carbin, S. Misailovic, M. Kling, and M. C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. 609–633.

[15] K.M. Chandy and C.V. Ramamoorthy. 1972. Rollback and Recovery Strategies for Computer Programs. *Computers, IEEE Transactions on* C-21, 6 (June 1972), 546–556.

[16] L. Degioanni, F. Risso, and G. Varenni. 2004. PCAP Next Generation Dump File Format. https://www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html. (March 2004).

[17] B. Demsky and A. Dash. 2008. Bristlecone: A Language for Robust Software Systems. In *Proceedings of the 22Nd European Conference on Object-Oriented Programming (ECOOP '08)*. 490–515.

[18] R. Fielding and J. Reschke. 2014. Hypertext transfer protocol (HTTP/1.1): Semantics and content. (2014).

[19] J. B. Goodenough. 1975. Exception Handling: Issues and a Proposed Notation. *Commun. ACM* 18, 12 (Dec. 1975), 683–696.

[20] J. Gray and A. Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.).

[21] E. Hamilton. 1992. JPEG File Interchange Format. (1992).

[22] S. Hanenberg. 2010. An Experiment About Static and Dynamic Type Systems: Doubts About the Positive Impact of Static Type Systems on Development Time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. 22–35.

[23] K. Huang and J. Wu E. B. Fernández. 1998. A Generalized Forward Recovery Checkpointing Scheme. In *IPPS/SPDP Workshops*. 623–643.

[24] J.F. Kurose and K.W. Ross. 2010. *Computer Networking: A Top-down Approach*.

[25] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. 1977. Abstraction Mechanisms in CLU. *Commun. ACM* 20, 8 (Aug. 1977), 564–576.

[26] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard. 2012. Automatic Input Rectification. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. 80–90.

[27] F. Long, S. Sidiroglou-Douskos, and M. Rinard. 2014. 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)*. 227–238.

[28] T. J. McCabe. 1976. A Complexity Measure. *IEEE Trans. Softw. Eng.* 2, 4 (July 1976), 308–320.

[29] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly, et al. 2010. The Genome Analysis Toolkit: a MapReduce framework for analyzing next-generation DNA sequencing data. *Genome research* 20, 9 (2010), 1297–1303.

[30] B. P. Miller, L. Fredriksen, and B. So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (Dec. 1990), 32–44.

[31] S. Nagarakatte, J. Zhao, M. M.K. Martin, and S. Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM '10)*. 31–40.

[32] S. Nanz and C. A. Furia. 2015. A Comparative Study of Programming Languages in Rosetta Code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. 778–788.

[33] G. C. Necula, S. McPeak, and W. Weimer. 2002. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. 128–139.

[34] V. Pankratius, F. Schmidt, and G. GarretÃşn. 2012. Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java. In *2012 34th International Conference on Software Engineering (ICSE)*. 123–133.

[35] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. 2005. Rx: Treating Bugs As Allergies—a Safe Method to Survive Software Failures. *SIGOPS Oper. Syst. Rev.* 39, 5 (Oct. 2005), 235–248.

[36] R Core Team. 2015. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org

[37] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. 155–165.

[38] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. 2004. 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)*. 21–21.

[39] O. Ruwase and M. S Lam. 2004. A Practical Dynamic Buffer Overflow Detector.. In *NDSS*, Vol. 2004. 159–169.

[40] Y. Shafranovich. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. https://tools.ietf.org/html/rfc4180. (Oct. 2005). RFC 4180.

[41] N. Shavit and D. Touitou. 1995. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (PODC '95)*. 204–213.

[42] J. Shen and M. Rinard. 2015. Filtered Iterators for Safe and Robust Programs in RIFL. http://hdl.handle.net/1721.1/100542. (2015). http://hdl.handle.net/1721.1/100542 MIT-CSAIL-TR-2015-036.

[43] A. Shinnar, D. Tarditi, M. Plesko, and B. Steensgaard. 2004. *Integrating support for undo with exception handling*. Technical Report. Microsoft Research. MSR-TR-2004-140.

[44] S. Sidiroglou and A. D. Keromytis. 2004. *Using Execution Transactions To Recover From Buffer Overflow Attacks*. Technical Report. Columbia University Computer Science Department. http://academiccommons.columbia.edu/item/ac:109823 CUCS–031–04.

[45] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. 2015. 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)*. 43–54.

[46] S. Siegel. 1956. *Nonparametric statistics for the behavioral sciences*.

[47] P. Stanley-Marbell and M. Rinard. 2015. Lax: Driver Interfaces for Approximate Sensor Device Access. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/stanley-marbell

[48] G. L. Steele, Jr. 1990. *Common LISP: The Language (2Nd Ed.)*.

[49] A. Stefik and S. Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *Trans. Comput. Educ.* 13, 4, Article 19 (Nov. 2013), 40 pages.

[50] Symantec Inc. 2005. *Symantec Internet security threat report: Vol. VII*. Technical Report.

[51] A. Warth, Y. Ohshima, T. Kaehler, and A. Kay. 2011. Worlds: Controlling the Scope of Side Effects. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. 179–203.

[52] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. 2003. Overview of the H.264/AVC Video Coding Standard. *IEEE Trans. Cir. and Sys. for Video Technol.* 13, 7 (July 2003), 560–576.

[53] F. Wilcoxon. [n. d.]. *Biometrics Bulletin* 6 ([n. d.]), 80–83.

[54] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*.