Detecting and Escaping Infinite Loops with Jolt

Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard

Massachusetts Institute of Technology, Cambridge, MA, USA
{mcarbin, misailo, mkling, rinard}@csail.mit.edu

Abstract. Infinite loops make applications unresponsive. Potential problems include lost work or output, denied access to application functionality, and a lack of responses to urgent events. We present *Jolt*, a novel system for dynamically detecting and escaping infinite loops. At the user's request, Jolt attaches to an application to monitor its progress. Specifically, Jolt records the program state at the start of each loop iteration. If two consecutive loop iterations produce the same state, Jolt reports to the user that the application is in an infinite loop. At the user's option, Jolt can then transfer control to a statement following the loop, thereby allowing the application to escape the infinite loop and ideally continue its productive execution. The goal is to enable the application to execute long enough to save any pending work, finish any in-progress computations, or respond to any urgent events.

We evaluated Jolt by applying it to detect and escape eight infinite loops in five benchmark applications. Jolt was able to detect seven of the eight infinite loops (the eighth changes the state on every iteration). We also evaluated the quality of escaping an infinite loop as an alternative to terminating the application. In all of our benchmark applications, escaping an infinite loop produced a more useful output than terminating the application. Finally, we evaluated how well escaping from an infinite loop approximated the correction that the developers later made to the application. For two out of our eight loops, escaping the infinite loop produced the same output as the fixed version of the application.

1 Introduction

```
From: "Armando Solar-Lezama" <asolar@csail.mit.edu>
To: "Martin Rinard" <rinard@csail.mit.edu>
Subject: Thanks

I was writing a document in Word this morning, and after about an hour of
unsaved work, Word went into an infinite loop that made the application
completely frozen. So, having listened to your talks too many times, I got
my debugger, paused the program, changed the program counter to a point a
few instructions past the end of the loop, and let it keep running from
there. Word went back to working as if nothing had ever happened. I was
able to finish my document, save it, and close Word without problems.

So thanks,
Armando.
```

As the above email illustrates, infinite loops can make an application unresponsive to its users. The potential consequences include loss of work or an inability to use the application for its intended purpose.

One potential solution (as deployed by Professor Solar-Lezama above) is to drop the application into a debugger, find the infinite loop, then move the program counter past the end of the loop, thereby enabling the application to continue its productive execution. Unfortunately, not everyone has the technical skill to use this solution. And even if one does, using the debugger, finding the loop, and moving the program counter past the end of the loop can be a tedious and annoying process.

1.1 Automatic Detecting and Escaping Infinite Loops

We present *Jolt*, a novel system for detecting and (if desired) escaping infinite loops. If a user suspects that an application may be in an infinite loop, he or she can instruct Jolt to monitor the execution of the application. Specifically, Jolt records the program state at the start of each loop iteration. The next time execution reaches the start of the loop, Jolt compares the current state to the saved state. If the current and saved states are the same, then the loop has made no progress and Jolt has detected an infinite loop. At the user's option, Jolt can escape the loop (i.e., transfers control to a statement after the loop to enable the application to continue its execution beyond the loop). The goal of the continued execution is to enable the application to save any pending work, finish any pending computation, or respond to any urgent events.

1.2 Evaluation

We evaluated Jolt by applying it to eight infinite loops in five applications (ctags, grep, indent, look, ping). We attached Jolt to each of these applications while they were executing on inputs that triggered the infinite loops. Jolt successfully detected seven of the eight infinite loops; the remaining loop changes the state on every iteration (Jolt is designed to detect only infinite loops in which the program state does not change across iterations).

As part of each case study, we used Jolt to exit the infinite loops and observed the resulting continued execution. In general, the applications are structured to process multiple input units (such as lines, modules, or records). The infinite loops occur when one of the input units hits a corner case in the application's code. Escaping the loop typically causes some perturbations in the computation on the current unit. But, by the time the application starts processing the next unit, it has recovered and is able to process this unit with no problems (unless, of course, this unit also triggers the infinite loop). The end result is that the application is often able to produce largely or even fully useful output.

We note that a similar phenomenon is partially responsible for the effectiveness of failure-oblivious computing [28] and SRS crash suppression [22] in enabling applications to recover from memory errors — because the applications tend to have short error propagation distances, errors that occur when processing one unit tend not to affect the processing of the next unit.

We also compared the output of escaping infinite loops with that of simply terminating the application at the infinite loop (for example, by hitting Ctrl-C). Terminating the application, of course, leaves it unable to process subsequent input units. And in some cases, the application produces no output at all — it is designed to produce all of its output after it has processed all of the input units. We found that for all of our applications, escaping the infinite loop produced a more useful output than terminating the application.

Finally, we acquired versions of the applications that were fixed by their developers. We then compared the outputs of escaping our infinite loops with the outputs of these versions of the applications. In two out of our eight infinite loops, escaping a loop produced an output that is identical to the output of the fixed version of the application. For the remaining infinite loops, output degradation was limited to the portion of the output that was generated from the input unit that caused the infinite loop.

1.3 Contributions

This paper makes the following contributions:

– **Detection and Escape:** It presents a system, Jolt, for detecting and (if desired) escaping infinite loops. Our technique uses both static source code instrumentation and dynamic binary instrumentation. Jolt statically instruments the source of an application with runtime calls that demarcate the entry, exit, and body of every loop in the control flow graph of each function in the program; Jolt does not instrument or detect loops due to recursion or unstructured control flow (i.e., exception handling).

  When instructed by a user, Jolt dynamically attaches to a running instance of the application and inserts instrumentation to record the state at the start of each loop iteration. As the application executes, Jolt compares the current state with the state from the previous iteration. If the states are equal, Jolt has detected an infinite loop. At the user's option, Jolt can then escape and continue execution at a statement following the loop.

– **Detection Evaluation:** It presents empirical results from applying Jolt to eight loops in five applications. Jolt detects seven of the eight loops (the remaining loop changes the state on every iteration). It also presents an evaluation of the performance of our technique; it imposes no more than 8.6% overhead on our applications when Jolt is not monitoring the application. And, when monitoring, Jolt detected all infinite loops in less than 1 second.

– **Escape Evaluation:** It presents empirical results that demonstrate that for all of our benchmark applications, escaping an infinite loop produces a more useful output than terminating the application. Moreover, escaping an infinite loop produces an output that is identical to the output of a manually fixed version of the application for two out of our eight infinite loops. In general, continued execution after the loop is successful because the applications tend to have short error propagation distances.

In our opinion, our results support the hypothesis that Jolt can provide a useful alternative to simply terminating the application when it encounters an infinite loop. We anticipate that Jolt will prove to be useful for interactive applications in which terminating the application would cause the user to lose work or leave the user without useful output. More generally, we expect that Jolt may also enable a wide range of applications to provide useful service even in the presence of infinite loops that would, in the absence of Jolt, render the application completely unresponsive.

2 The Jolt System

To provide users with a low-overhead system for infinite loop detection and escape, we have designed Jolt around two components:

Compiler: Jolt's compiler enables a developer or user to compile the source code of his or her application to obtain a binary executable that is amenable to infinite loop detection. In particular, Jolt's compiler adds lightweight instrumentation to the source of the application to identify the boundaries of loops, which can be difficult to identify accurately from a binary executable [15, 34].

Detector: Jolt's detector can, at the user's request, dynamically attach to and analyze a running instance of an application that the user believes is caught in an infinite loop (if the application has been compiled with Jolt's compiler). If the detector determines that the application is caught in an infinite loop, it presents the user with the option to escape the loop.

2.1 Example

To illustrate how Jolt compiles and analyzes an application, we present an in-depth example of applying Jolt to an infinite loop in ctags, one of our benchmark applications.

Ctags scans program source files to produce an index that maps program entities (e.g., modules, functions, and variables) to their line numbers within the source files [1]. Ctags contains multiple modules for parsing and extracting the index, each of which is specific to a particular programming language. An integrated development environment can later use such an index file to allow programmers to quickly navigate to the definitions of modules, functions, and other program entities by name.

```
def get_pkgdocs(self):
  if symbols:
    retstr += """\n\nGlobal symbols from subpackages""" \
    """\n--------------------------------\n""" + \
    self._format_titles(symbols,'-->')
```

Fig. 1. Example Python Code

Figure 1 presents a Python code snippet taken from the numpy numerical matrix manipulation routine library. Ctags was designed to parse this source code and output an index, which indicates that, e.g., the function `get_pkgdocs()` begins on Line 1.

This code snippet uses multi-line strings (which are delimited by matched pairs of triple-quote literals, `'''` or `"""`, and can span more than one line) on Lines 3 and 4 to construct the string `retstr`. The backslash between the two lines is admissible Python syntax and appears in the original file; in Python two lines that are separated by a backslash are treated as a single line. As a consequence, ctags merges the two lines into a single line during its preprocessing stage. However, when multiple multi-line strings appear on the same line in a Python source file, ctags version 5.7beta can enter an infinite loop.

2.2 Infinite Loop

```
 1  static void find_triple_end(char const *string, char const **which) {
 2    char const *s = string;
 3    while (1) {
 4      s = strstr (string, *which);
 5      if (!s) break;
 6      s += 3;
 7      *which = NULL;
 8      s = find_triple_start(s, which);
 9      if (!s) break;
10      s += 3;
11    }
12  }
```

Fig. 2. Source Code for Ctags

Figure 2 presents `find_triple_end()`, the function from Ctags's Python module that loops infinitely on the code snippet from Figure 1. The function serves to identify if `string`, which points to a character buffer containing a single line of text from a parsed file, closes an already open multi-line string. The parameter `which` contains the delimiter that began the multi-line string (either `'''` or `"""`).

At the beginning of each iteration of the loop, `s` points to some position in `string` and `which` contains the triple-quote that began the last multi-line string. Within the loop, if `s` does not contain a matching triple-quote, then the loop exits (Line 5). If `s` does contain a matching triple-quote, then the computation 1) records that the currently opened multi-line string has been closed, by setting `which` to `NULL` on Line 7, and 2) checks if `s` contains any additional triple-quotes.

If `s` does not contain an additional triple-quote, then the computation exits the loop (Line 9). Otherwise, the computation 1) records that a new multi-line string has been opened (by updating `which` in `find_triple_string()`), and 2) updates `s` to point to the character after the newly found triple-quote. The

```c
#define LOOP_ID 148

static void find_triple_end(char const *string, char const **which) {
  char const *s = string;

  jolt_loop_entry(LOOP_ID);
  while (1) {
    if (!jolt_loop_body(LOOP_ID)) {
      goto jolt_escape;
    }
    s = strstr (string, *which);
    if (!s) {
      jolt_loop_exit(LOOP_ID);
      break;
    }
    s += 3;
    *which = NULL;
    s = find_triple_start(s, which);
    if (!s) {
      jolt_loop_exit(LOOP_ID);
      break;
    }
    s += 3;
  }
jolt_escape:
}
```

Fig. 3. Instrumented Source Code for Ctags

computation then returns to the beginning of the loop to look for a triple-quote that closes the newly opened multi-line string.

The programmer wrote this loop with the intention that each iteration of the loop would start at some position in `string` (given by `s`) and either exit, or continue with another iteration that starts at a later position in `string`. To establish this, the value of `s` is incremented by the functions `strstr()` (Figure 2, Line 4) and `find_triple_string()` (Figure 2, Line 8).

However, in the call to `strstr()` the developer mistakenly passed `string`, instead of `s`, as the starting position for each iteration. As a consequence, every iteration of the loop starts over at the beginning of `string`, which can cause an infinite loop. For example, if the triple-quotes of the first and the second multi-line string are of the same type (as in Figure 1), then at the beginning of every loop iteration (except the first), the values of `s` and `which` are always the same: `s` equals to the starting position of the second multi-line string and `which` contains the triple-quote that starts the second multi-line string.

2.3 Compilation

Figure 3 presents the instrumentation that Jolt's compiler adds to the source code of this loop in ctags. For every loop in the application, Jolt identifies and marks the following:

- **Loop ID:** Jolt gives each loop in the application a unique identifier (Line 1).
- **Loop Entry:** At the entry point of the loop, Jolt adds a call to the function `jolt_loop_entry()` to notify Jolt's runtime that the application has reached the beginning of a loop (Line 6).
- **Loop Exit:** At each exit point from the loop, Jolt adds a call to the function `jolt_loop_exit()` immediately before exiting the loop to notify Jolt's runtime that the application is about to exit a loop (Lines 13 and 20).
- **Loop Body and Loop Escape Edge:** Jolt adds a call to the function `jolt_loop_body()` at the start of the loop body to let Jolt control the execution of the loop (Line 8). If `jolt_loop_body()` returns true, then the application will execute the body of the loop. If `jolt_loop_body()` returns false, then the application will escape the loop by branching to the block immediately after the loop, which is marked by the label `jolt_escape` (Line 25). By default, `jolt_loop_body()` returns true if a user has not used Jolt's detector to attach to the application.

After instrumenting ctags source code, Jolt uses the LLVM 2.8 compiler infrastructure [16] to compile the source code down to an executable (a 32-bit or 64-bit ELF executable in our current implementation). Though the instrumented executable incurs some overhead (Section 6), its semantics are exactly the same as that of the uninstrumented application — that is, until a user instructs Jolt's detector to attach to a running instance of the application.

2.4 Detection

Once the user believes that ctags may be caught in an infinite loop, he or she can use Jolt's user interface to scan the list of active system processes and select the suspect ctags process. When the user selects the process, Jolt's infinite loop detector attaches to the running process and begins monitoring its execution.

Conceptually, Jolt records a snapshot of the state of the application at the beginning of each loop iteration. If ctags is caught in the infinite loop from Section 2.2, Jolt's detector will recognize that 1) the application modifies only the variables `s` and `which`, and 2) that these variables have the same values at the beginning of each loop iteration. Given this observation, Jolt will report to the user that the application has entered an infinite loop.

2.5 User Interaction

After Jolt detects an infinite loop, it presents the user with the option to escape the loop. If the user chooses to escape the loop, he or she can place Jolt into one of two interaction modes:

– **Interactive Mode:** After Jolt forces the application to escape the loop, Jolt detaches from the application. Jolt will not detect any subsequent infinite loops unless the user again instructs Jolt to attach to the running application.
– **Vigilant Mode:** After Jolt forces the application to escape the loop, Jolt stays attached to the application. Jolt will continue to detect and escape infinite loops without further user interaction. Vigilant mode is useful when the application encounters an input that repeatedly elicits infinite loops.

It is also possible to support additional modes in which Jolt stays attached, but asks the user each time it detects an infinite loop before escaping the loop. Or, if Jolt is unable to detect an infinite loop, a user may, at his or her own discretion, choose to escape a loop that has been executing for a long time.

2.6 Escaping the Infinite Loop

Terminating ctags during the infinite loop from Section 2.2 would cause the user to lose some or all of the indexing information for the current file. Moreover, terminating ctags would leave it unable to process any subsequent files that could have been passed on the command line. If, instead, a user elects to escape the loop, then Jolt will force the application to exit the loop by returning `false` for the next call to `jolt_loop_body()`. As a consequence, ctags will terminate and produce a well-formed output. This output will include some of the definitions from the current file and all of the definitions from any subsequent files.

The quality of the output from the current file depends on the position of the triple-quote that closes the second string. If the triple-quote is on the same line (such as in Figure 1), then the quotes become unmatched, effectively causing ctags to treat the remainder of the current file as a multi-line string. On the other hand, if the triple-quote is on a subsequent line, then ctags will produce the exact same set of definitions as intended by the developers (which we verified by inspecting a later, fixed version of the application).

3 Implementation

Our design adopts a hybrid instrumentation approach that uses both static source code instrumentation and dynamic binary instrumentation. Jolt statically inserts lightweight instrumentation into the application to monitor the application's control flow. Then, after it has attached to the application, Jolt inserts heavyweight dynamic binary instrumentation to monitor changes in the application's state. Between these two components, our design balances Jolt's need for precise information about the structure of an application with our desire to minimize overhead on the application when it is not being monitored.

3.1 Static Instrumentor

Jolt's static instrumentor provides Jolt's detector with control flow information that may otherwise be difficult to extract accurately from the compiled binary of a program. The static instrumentor inserts function calls that notify the detector of the entry, body, and exit of each natural loop [21] in the control flow graph of each function in the program (as we presented in Section 2). Jolt currently does not instrument loops that occur due to recursion or unstructured control flow, such as exception handling or gotos that produce unnatural loops.

Jolt's static instrumentor also selects an escape destination for each loop in the application. The static instrumentor chooses an escape destination from one of the normal exit destinations of the loop. In general, a loop may contain multiple exit destinations (this can occur, for example, if the loop body uses goto statements to exit the loop). Jolt currently chooses the first loop exit as identified by LLVM.

It also possible for the loop to contain no exits at all. This can happen, for example, if the program uses an exception mechanism such as `setjmp/longjmp` to exit the loop. In this case Jolt inserts a return block that causes the application to return out of the current procedure. When Jolt escapes the infinite loop, it transfers control to this return block. Researchers have demonstrated that simply returning from a function can be an effective way to work around an error within its computation [30].

We have implemented the static instrumentor as an LLVM compiler pass that operates on LLVM bitcode, a language-independent intermediate representation. Given the instrumented bitcode of an application, we then use LLVM's native compiler to generate a binary executable.

3.2 Dynamic Instrumentor

When a user enables Jolt's infinite loop detection on a running program, Jolt's dynamic binary instrumentation component dynamically attaches the running program and inserts instrumentation code to record the state of the program as it executes. Jolt's instrumentation (conceptually) records, at the top of each loop, a snapshot of the state that the last loop iteration produced. To avoid having to record the entire live state of the application, Jolt instruments the application to produce a *write trace*, which captures the set of registers and addresses that each loop iteration writes.

Write Trace: Jolt instruments each instruction in the application that modifies the contents of a register or memory address. For each register or memory address that an instruction modifies, the instrumentation code dynamically records either the identifier of the register or the memory address into the write trace.

Snapshot: At the top of each loop, the inserted Jolt instrumentation uses the resulting write trace to record a snapshot. This snapshot contains 1) the list of registers and memory addresses written by the last iteration and 2) the values in those registers and memory addresses at the end of the last iteration. Jolt records a snapshot only if it has a complete write trace from the last loop iteration (the trace may be incomplete if the user attached Jolt to the application sometime during the iteration).

Library Routine Abstraction: To record a full write trace of an application, Jolt must instrument all of the application's running code, including libraries. However, some libraries may modify internal state that is unobservable to the application. For example, some libc routines modify internal counters, (i.e., the number of bytes written to a file, or the number of memory blocks allocated by the program), that change after every invocation of the routine. If an application invokes one of these routines during a loop that is, otherwise, producing the same state on each iteration, then Jolt will be unable to detect the infinite loop. However, these counters are often either 1) not exposed to the application, or 2) exposed but not used by the application in a given loop. Therefore, we allow Jolt to accept a set of *library routine abstractions* to explicitly specify the set of observable side-effects of library routines.

A library routine abstraction specifies if the routine modifies observable properties of its arguments. For example, consider the `write` routine from libc:

```
ssize_t write(int filedes, const void *buf, size_t nbyte);
```

This function does not modify the contents of `buf`, but it does modify the current position of the file cursor, which the application can query by calling `ftell(filedes)`. If during an infinite loop, Jolt does not observe any calls from the application to `ftell(filedes)`, then Jolt can exclude the side-effects of a call to `write(filedes, ...)` from the snapshot.

We have implemented library routine abstractions for the subset of libc library calls that are invoked by our benchmark programs (e.g., `read`, `write`, `printf`). We anticipate that library routine abstractions need only be implemented for libraries that are considered a part of the runtime system of the application (e.g., allocation, garbage collection, and input/output routines).

Detection: At the beginning of each loop iteration, Jolt's detector compares the snapshots of the two previously executed loop iterations. If the two snapshots are the same — i.e., both snapshots contain the same registers and memory addresses in their write traces and the recorded values for these registers and memory addresses in the snapshots are the same — then Jolt reports that it has detected an infinite loop.

We have implemented the dynamic instrumentor on top of the Pin dynamic program analysis and instrumentation framework [18]. Our use of Pin enables the dynamic instrumentor to analyze both Linux and Windows binaries that have been compiled for the x86, x64, or IA-64 architectures.

4 Empirical Evaluation

In this section, we present a set of case studies designed to evaluate how well Jolt enables users to detect and escape infinite loops in real applications.

Benchmark	Version	Reference Version	Bug Report	Location
ctags-fortran	5.5	5.5.1	Ctags-734933	`fortran.c, parseProgramUnit, 1931`
ctags-python	5.7b (646)	5.7b (668)	Ctags-1988027	`python.c, find_triple_end, 364`
grep-color	2.5	2.5.3	gnu.utils.bugs 03/21/2002 message 9	`grep.c, prline, 579`
grep-color-case				`grep.c, prline, 562`
grep-match				`grep.c, prline, 532`
ping	20100214	20101006	CVE-2010-2529	`ping.c, pr_options, 984`
look	1.1 (svr 4)	-	[37]	`look.c, getword, 172`
indent	1.9.1	2.2.10	[37]	`indent.c, indent, 1350`

Table 1. Studied Infinite Loops

4.1 Benchmarks

Table 1 presents the loops that we use in our evaluation. The first column (Benchmark) presents the name we use to refer to the loop. The second column (Version) presents the version of the application with the infinite loop. The third column (Reference Version) presents the version of the application in which the infinite loop has been corrected. The fourth column (Bug Report) presents the source of the infinite loop bug report. The fifth column (Location) presents the file, the function, and the line number of the infinite loop.

We evaluated Jolt on eight loops in five benchmark applications. We selected applications for which 1) bug reports of infinite loops were available, 2) we could reproduce the reported infinite loops, and 3) we could qualitatively characterize the effect of escaping the loop on the application's output. All these applications are commonly used utilities that the user either invokes directly, from the command line, or as a part of a larger workflow:

- **ctags:** Scans program source files to produce an index that maps program entities (e.g., modules, functions, and variables) to their locations within the source files [1]. We investigate two infinite loops in ctags:
  - **ctags-fortran:** The ctags Fortran module (version 5.5) has an infinite loop that occurs when processing 1) source code files with variable and type declarations separated by a semicolon, or 2) syntactically invalid source files with improperly nested components. In both cases, ctags enters a mode in which it infinitely loops when it is unable to recognize certain valid Fortran keywords.
  - **ctags-python:** The ctags Python module (version 5.7 beta, svn commit 646) has an infinite loop that occurs when one multi-line string literal ends on a line and another multi-line string literal starts on the same line (as we discussed in Section 2.1).
- **grep:** Matches regular expressions against lines of text within a single input file or multiple input files [2]. We investigate three infinite loops in grep version 2.5. Although all of these loops are distinct, they appear to share a common origin via a copy/paste/edit development history.

  - **grep-color:** This infinite loop occurs when grep is configured to display matching parts of each line in color and is given a regular expression with zero-length matches.
  - **grep-color-case:** This infinite loop occurs when grep is configured to display matching parts of each line in color with case-insensitive matching and is given a regular expression with zero-length matches.
  - **grep-match:** This infinite loop occurs when grep is configured to print only the parts of each line that match the regular expression and is given a regular expression with zero-length matches.

- **ping:** Ping client is a computer network utility which checks for the reachability of a remote computer using the Internet Control Message Protocol (ICMP) echo messages. The infinite loop can occur when processing certain optional headers (time stamps and trace route records) of the echo reply message from the remote computer.
- **look:** Prints all words from a dictionary that have the input word as a prefix. The infinite loop occurs when look's binary search computation visits the last entry in the dictionary file and this last entry is not terminated by a newline character. We were not able to obtain the reference version of look, but instead manually fixed the application to produce a correct result (according to our understanding of its functionality).
- **indent:** Parses and then formats C and C++ source code according to a specified style guideline [3]. This infinite loop occurs when 1) the input contains a C input preprocessor directive on the last line of the input, 2) this line contains a comment, and 3) there is no end of line character at the end of this last line.

4.2 Methodology

For each of our benchmark loops, we performed the following tasks:

- **Reproduction:** We obtained at least one input that elicits the infinite loop, typically from the bug report. Where appropriate, we constructed more inputs that cause the application to loop infinitely.
- **Loop Characterization:** We identified the conditions under which the infinite loop occurs. This includes distinctive properties of the inputs that elicit the infinite loop and characteristics of the program state. We also characterized the execution behavior (e.g., resource consumption and output) of the application during the infinite looping.
- **Infinite Loop Detection:** We first compiled the application with Jolt's compiler to produce an instrumented executable. We then ran the executable on our eliciting inputs to set the application into an infinite loop. Finally, we dynamically attached Jolt's detector to the running application to determine if Jolt could detect the infinite loop.

– **Effects of Escaping Infinite Loop:** We characterized the internal behavior of the application after using Jolt to escape the loop, including the effects of the escape on the output and the memory safety of the application. We used manual inspection and testing to ensure that the output of the application is well-formed, and Valgrind [23], to determine if the continued execution performed any invalid memory operations (such as out of bounds accesses or memory leaks).

– **Comparison with Termination:** One common strategy for dealing with an application that is in an infinite loop is to simply terminate the application. We compared the output that we obtain from terminating the application to the output from the version that uses Jolt to escape the infinite loop. Specifically, we investigated whether using Jolt helped produce a more useful output than terminating the application.

– **Comparison with Manual Fix:** We evaluated how well escaping from an infinite loop approximated the correction that the developers later made to the application. We obtained a version of the application in which the infinite loop had been manually corrected. When then compared the output from escaping the loop to the output from the fixed version of the application. Specifically, we investigated the extent to which the output produced by the application after using Jolt matched the output of the manually fixed application.

4.3 Results

Table 2 summarizes the results of our evaluation of Jolt as a technique for detecting and escaping infinite loops. The first column (Benchmark) presents the infinite loop name. The second column (Detection) presents whether Jolt successfully detected the infinite loop. If an entry in this column contains the symbol ●, detection succeeded; if it contains ○, then detection failed — we use the same notation for positive and negative results in each subsequent column.

The third column (Sanity Check) presents whether escaping the loop maintained the memory consistency, as reported by Valgrind. The fourth column (Comparison with Termination) presents whether using Jolt to escape the infinite loop produces a more useful output than the output that we obtain after terminating the application. Finally, the fifth column (Comparison with Manual Fix) presents whether using Jolt to escape the infinite loop produces the same output as the reference version for every input to the application. If an entry in this column contains the symbol ◐, then the outputs are the same for some, but not all, inputs.

For infinite loops that Jolt failed to detect, we still present results that describe the behavior of the application after escaping the loop. We performed these experiments by modifying Jolt to escape the loop, even though it had not detected an infinite loop.

Benchmark	Detection	Sanity Check	Comparison With Termination	Comparison With Manual Fix
ctags-fortran	●	●	●	◐
ctags-python	●	●	●	◐
grep-color	●	●	●	◐
grep-color-case	●	●	●	◐
grep-match	●	●	●	◐
ping	●	●	●	●
look	●	●	●	●
indent	○	●	●	●

Table 2. Summary Results for Infinite Loops

Infinite Loop Detection: Jolt was able to detect seven out of eight infinite loops in our benchmark applications. For these infinite loops, Jolt identified that the state of the program remained the same in adjacent loop iterations, and escaped the loop immediately. Jolt failed to detect the infinite loop in indent because the state changed on every iteration through the loop. We discuss the reasons for why Jolt failed to detect this infinite loop in Section 5.2

Sanity Check: For all of our benchmarks the resulting continued execution of the application exhibited no memory errors.

Comparison with Termination: For all our benchmarks, our evaluation indicates that using Jolt to escape the loop resulted in outputs that contain as much or more useful information than the outputs obtained by terminating the application. Terminating the applications after encountering an infinite loop left the application unable to process subsequent input units (files, lines or requests). For ctags and indent (when processing multiple input files), grep, ping, and look, terminating the application produced outputs only up to the point of termination (and none thereafter). Ctags and indent (when operating on a single input file with, potentially, multiple lines) are designed to produce their outputs at the end of the computation. Therefore, terminating the application did not yield any output at all. As an extreme example, terminating indent while in the infinite loop caused it to overwrite the input source code file with an empty file. Escaping the infinite loop with Jolt, on the other hand, not only helped the application finish processing the current input, but also enabled it to continue to successfully process subsequent inputs.

Comparison with Manual Fix: For ping, look and indent, the outputs of the application for which we applied Jolt, and the outputs of the application with a manually fixed bug were identical. The computations in these loops finished processing the entire input before the loop started infinitely looping.

Applying Jolt to infinite loops in ctags and grep helped produce an output containing a part of the output of the manually corrected application. In Section 2.6 and Section 5.1, we present a more detailed characterization of the quality of these outputs.

5 Selected Case Studies

In Section 2, we presented an extended case study of ctags-python that demonstrated Jolt's overall approach to infinite loop detection and escape. In this section we now present two additional case studies that demonstrate the main characteristics of the infinite loops that we analyzed and the details of our evaluation. In particular, these case studies highlight the utility of vigilant mode (Section 2.5), the utility of library abstraction (Section 3.2), and some of the limitations of the Jolt's detector. We have made detailed case studies of the rest of our benchmark applications available online at http://groups.csail.mit.edu/pac/jolt.

5.1 Grep

Figure 4 presents the source code of the grep-color loop, which colors the part of the current line matching a regular expression. Grep executes this loop when the user provides the `--color` flag on the command line. This loop, along with the other two infinite loops in grep, occur in the function `prline`, which is responsible for presenting the text that matches the regular expression to the user. The other two infinite loops have the same infinite loop behavior and a similar structure.

```
 1  while ( (match_offset = (*execute) (beg, lim - beg, &match_size, 1))
 2          != (size_t) -1)
 3  {
 4    char const *b = beg + match_offset;
 5    /* Avoid matching the empty line at the end of the buffer. */
 6    if (b == lim)
 7      break;
 8    fwrite (beg, sizeof (char), match_offset, stdout);
 9    printf ("\33[%sm", grep_color);
10    fwrite (b, sizeof (char), match_size, stdout);
11    fputs ("\33[00m", stdout);
12    beg = b + match_size;
13  }
```

Fig. 4. Source Code for Grep-color Infinite Loop

Infinite Loop: The computation stores the pointer to the current location on the line in the variable `beg`. The function `execute()` on Line 1 searches for the next match starting from the position `beg`. Each time a match on the current line is found, this pointer is incremented to advance the search, first by adding the offset to the position of the next match (`match_offset`; Line 4), and then by adding the size of the match (`match_size`; Line 12). However, when using a regular expression that matches zero length strings (such as `[0-9]*`), the variable `match_size` will have value zero. Consequently, the value of pointer `beg` will not increase past the current match and the progress of the loop execution will stop.

The loop can still output the first non-zero length match at the beginning of the line, since grep uses a greedy matching strategy (it selects the longest string that matches the pattern). For example, for the input `echo "1 one" | ./grep "[0-9]*" --color`, the output contains a colored number `1`, but following loop iterations do not progress past this point — the string `one` is never printed. On the other hand, grep-match will output a single newline character (`'\n'`) for each iteration as it loops after the first match. In the previous example, it will output a number of newline characters after matching `1`.

Infinite Loop Detection: While in the infinite loop, the computation outputs non-printable characters (which control the text color) to the standard output stream in every iteration. The printing does not influence the termination of the loop, but may change internal counters and output buffer pointers within the standard library, which are not observable by the application, but would prevent Jolt from detecting the infinite loop. Thus, we apply library routine abstraction, which we described in Section 3.2 to allow Jolt disregard possible changes of the internal state of the library routines and enable detecting this infinite loop.

Effects of Escaping Infinite Loop: Applying Jolt to grep when it has entered an infinite looping state escapes the current loop (which also halts printing newline characters if the loop was doing so). The remainder of the current line is skipped. If Jolt operates in vigilant mode, grep will not print numerous spurious newline characters in grep-match case because Jolt escapes the loop after only two iterations, printing only one additional newline character. If Jolt operates in interactive mode, grep will print a number of newline characters for each line before the user instructs Jolt to terminate the loop causing the application to proceed to the next line.

For grep-color and grep-color-case loops, applying Jolt allows all matching lines of the input to be displayed. However, on a given line that contains multiple matches, only the first match will be colored. For example, for the sample input `echo "1 one 1" | grep -E "[0-9]*" --color`, grep outputs the desired line ('`1 one 1`'), but only the first "1" is colored. Using the `-o` command line flag to print only the matching string, grep outputs only the first match on each line, followed by newline characters until user invokes Jolt. For example, for the input `echo "1 one 1" | grep "[0-9]*" -o`, grep outputs a single line, containing a "1" (unlike two lines with value "1" that a corrected version of the application generates). Escaping the loop after printing the first match "1" skips the remainder of the line, which contains the second match "1".

Comparison with Termination: Terminating the execution of grep causes it to not process any line after the first zero-length match (which is effectively any line of the input). In contrast, using Jolt allows grep to continue searching for matches on subsequent lines in the input.

Comparison with Manual Fix: The correction that the application developers applied for the three infinite loops in Version 2.5.3 causes the application to continue printing the line even after encountering the match of length zero. As part of the fix, the developers completely restructured the control flow, and removed the loops in the progress. This version of the application prints correctly all non-zero matches, and skips zero-length matches.

While in Section 4 we presented the results of our comparison with Version 2.5.3, we also analyzed the correction that the developers of grep implemented in Version 2.5.1. This fix was in place for three years before the release of 2.5.3. In this version, the developers added the code `if (match_size==0) break;` before Line 8 to exit the loop when encountering a zero-length match. The effect of this manual fix is the same as using Jolt to escape the loop.

5.2 Indent

Figure 5 presents the simplified version of the loop that handles comments that occur within or on the same line after a preprocessor directive in C programs.

```
1   while (*buf_ptr != EOL || (in_comment && !had_eof)) {
2
3     if (e_lab >= capacity_lab) e_lab = extend_lab()
4
5     *e_lab = *buf_ptr++;
6     if (buf_ptr >= buf_end) buf_ptr = fill_buffer (&had_eof);
7
8     switch (*e_lab++) {
9     case ’\’:
10       handle_backslash(&e_lab, &buf_ptr, &in_comment); break;
11    case ’/’:
12       handle_slash(&e_lab, &buf_ptr, &in_comment); break;
13    case ’"’:
14    case ’´’:
15      handle_quote(&e_lab, &buf_ptr, &in_comment); break;
16    case ’*’:
17      handle_asterisk(&e_lab, &buf_ptr, &in_comment); break;
18    }
19  }
```

Fig. 5. Source Code for Indent Infinite Loop

Infinite Loop: The loop reads the text from the input buffer (pointed to by `buf_ptr`), formats it and appends it to the output buffer (pointed to by `e_lab`). The function `extend_lab()` on Line 3 increases the size of the output memory buffer if needed by using a library function `realloc()`. The function `fill_buffer()` on Line 6 reads a single line from the input file to the input buffer. If this function reads past the input file, it writes a single character ‘`\0`’ to the input buffer and sets the `had_eof` flag. Finally, the loop body recognizes the comment’s start and end characters, and sets `in_comment` appropriately.

The analysis of the loop condition on Line 1 shows that the loop computation ends only if 1) the input line contains the newline character and it is not in the comment (`*buf_ptr == EOL && !in_comment`), or 2) if the input line contains the newline character and it has reached the end of file (`*buf_ptr == EOL && had_eof`). The loop condition does not account for the case when loop reads the entire input file, but the last line does not end with the newline character (the value of `buf_ptr` in this case is equal to `'\0'`).

Infinite Loop Detection: While in the infinite loop, each iteration appends a spurious `'\0'` character to the output buffer, and the capacity of the output buffer is occasionally increased. Eventually, the output buffer can consume all application memory and cause the application to crash. Note that this update of the output buffer is the reason Jolt in its current version cannot detect this loop as infinite.

Effects of Escaping Infinite Loop: Although Jolt cannot detect the infinite loop in this application, we manually instructed Jolt to escape the loop to investigate its effect. After escaping the loop using Jolt, the application terminated normally, producing a correctly indented output file. Note that this infinite loop only happens after indent has processed all of the input file; the only remaining task at this point is to copy the output buffer to a file. Escaping the loop enables the application to proceed on to correctly execute this task.

Comparison with Termination: Terminating the application when it enters the infinite loop prevents the application from executing the code to print the output buffer to the file. Because the default configuration of indent overwrites the input file, the user is left with an empty input file. Terminating the application also causes indent to skip processing any subsequent files. Escaping the loop, on the other hand, produces the correct output for the file that elicits the infinite loop, and all remaining files.

Comparison with Manual Fix: The developer fix of the loop in Version 2.2.10 modifies a condition on Line 1 to test `has_eof` flag whether the input file has reached the end, before checking for the newline character in the input buffer. Escaping the infinite loop causes the application that used Jolt to produce the same result as the reference application version.

6 Performance

In this section, we present performance measurements designed to characterize Jolt's instrumentation overhead in normal use (Section 6.1) and the time required for detection of infinite loops in our benchmark applications (Section 6.2). We performed our performance measurement experiments on an 8-core 3.0GHz Intel Xeon X5365 with 20GB of RAM running Ubuntu 10.04.

6.1 Instrumentation Overhead

We designed this experiment to measure the overhead of adding instrumentation code to track the entry, exit, and body of each loop in the application (as described in Section 2). We measured the overhead of Jolt's instrumentation in normal use by running each application, with and without instrumentation, on inputs that do not cause infinite loops. In this experiment, infinite loop detection is never deployed.

Benchmarks: Our benchmark suite consists of the following workloads:

- **ctags-fortran:** we crafted five workloads by executing ctags version 5.5 with five different command line configurations on the Fortran language files of scipy, a suite of scientific libraries for the Python programming language [7]. The source code of these programs totals 81800 lines of code.
- **ctags-python:** we crafted five workloads by executing ctags version 5.7b (646) with five different command line configurations on the Python language files of numpy, a library of numerical matrix manipulation routines for Python [6]. The source code of these programs totals 72218 lines of code.
- **grep:** we crafted five workloads by executing grep version 2.5 with five different regular expressions on the concatenated C source code of grep, gstreamer[5], and sed[4]. We crafted regular expressions designed to match elements within C source code; namely, strings, comments, primitive data types (e.g., int, long, or double), parenthesized expressions, and assignment statements. The source code of these programs totals 35801 lines of code.
- **ping:** we crafted five workloads by executing ping client with different options, including targeting a remote machine on a local network (the same machine we used to reproduce the infinite loop), and the local host. We ran the same server on the remote machine that we used to elicit the infinite loop. For each ping execution we send multiple requests to the server (in particular, 100 requests to the remote host and 1,000,000 to the local host), without delay between the requests.
- **look:** we crafted five workloads by executing look version 1.1 (svr 4) with five query words and a corpus of 98569 words from the American English dictionary supplied with Ubuntu 10.04.
- **indent:** we crafted five workloads by executing indent version 1.9.1 with five different indentation styles on the C source code of gstreamer (15608 lines of code).

Methodology: To evaluate the instrumentation overhead for a single workload, we first ran the workload five times without measurement to warm the system's file cache (and, thus, overestimate the impact of instrumentation by minimizing I/O time). We then measured the execution time of each workload twenty times across two configurations: ten times to measure the execution time of the uninstrumented application and ten times to measure the execution time of the instrumented application.

To compute the instrumentation overhead (i.e., a slowdown) for a single workload, we take the median execution time of the ten executions of the instrumented application over the median execution time of the ten executions of the uninstrumented application. We use the median to filter out executions — both slow and fast — that may be outliers due to performance variations in the execution environment.

Benchmark	**Average**	**Lowest**	**Highest**
ctags-fortran	1.073	1.068	1.080
ctags-python	1.052	1.035	1.057
grep	1.025	1.014	1.028
ping	1.016	1.005	1.024
look	1.0	1.0	1.0
indent	1.084	1.082	1.086

Table 3. Performance Overhead of the Instrumentation

Results: Table 3 presents the results of the instrumentation overhead measurement experiments. The first column in Table 3 (Benchmark) presents the name of the benchmark. The second column (Average) presents the weighted average of the slowdowns over each benchmark's five workloads. The third column (Lowest) presents the lowest slowdown that we observed over each benchmark's five workloads. The fourth column (Highest) presents the highest slowdown that we observed over each benchmark's five workloads.

Jolt's overhead varies between 0.5% (the lowest observed overhead for ping) and 8.6% (the highest observed overhead for indent). In our experiments we found that the overhead imposed by Jolt on look was, in practice, too small to reliably distinguish it from the noise of the benchmark environment. We also note that the results for ping depend on the status of the network and the physical distance between the hosts. While we used short physical distances between the hosts and no delay between the requests to decrease the network variability and to account for the worst case, we expect that in a typical use the communication time will dominate the processing time, making the overhead minimal.

6.2 Infinite Loop Detection

We designed this experiment to evaluate how quickly Jolt can detect an infinite loop in a running application.

Methodology: To perform this experiment, we ran each infinite loop from our case studies on an input that elicits an infinite loop and then attached Jolt. We then allowed Jolt to run for two seconds; if Jolt did not detect the loop within two seconds, then we classified the loop as undetectable.

For each detected infinite loop in our case studies, we gathered 1) the time required for Jolt to detect the infinite loop, 2) the footprint, in number of bytes, of each infinite loop iteration, and 3) the length, in number of instructions, of

Benchmark	Time (s)	Footprint (bytes)	Length
ctags-fortran	0.319	240	256
ctags-python	0.334	312	992
grep-color	0.585	992	4030
grep-color-case	0.579	992	4036
grep-match	0.490	846	2506
ping	0.287	192	54
look	0.296	300	378

Table 4. Infinite Loop Detection Statistics for Benchmark Applications

each infinite loop. Each of these numbers corresponds to the second, third, and fourth columns of Table 4, respectively:

– **Time:** To measure the detection time, we repeatedly (five times) measured the absolute time that elapsed from the instant when Jolt attached to the application until the instant when Jolt detected that the loop iterations do not change the state. We report the median detection time over these trials.
– **Footprint:** We measured the memory footprint of the infinite loop by recording the number of bytes of the program state that Jolt recorded in the snapshot at the beginning of each iteration of the infinite loop. As discussed in Section 3, Jolt records the value of a register or memory address at the beginning of the loop only if it was written during the execution of the loop.
– **Length:** We measured the length of the loop by recording the number of instructions dynamically executed during one iteration of the loop. Our reported numbers count only user-mode instructions that wrote to a register or a memory location — this, therefore, excludes instructions executed by the operating system kernel and nop instructions that do not modify the state of the application.

Results: Table 4 presents the results of our infinite loop detection experiment. The times required to detect an infinite loop are all less than 1 second and the footprint of each infinite loop is less than 1 KB. Given that ping's infinite loop, our smallest benchmark loop (by number of instructions), takes Jolt 0.287 seconds to detect, our infinite loop detection technique is predominantly bounded from below by the time required to initialize the Pin instrumentation framework.

7 Limitations

Jolt currently detects only infinite loops that do not change program state between iterations. In general, infinite loops can change state between iterations, or cycle between multiple recurring states. We anticipate that Jolt's detector could be extended to eliminate changing state that does not affect a loop's termination condition, track multiple states, or use symbolic reasoning to prove non-termination [14, 35, 9].

Jolt does not consider the effects of multiple threads on the termination of the application. For example, Jolt may incorrectly report that an application is in an infinite loop if the application uses ad-hoc synchronization primitives (e.g., spin loops) [38]. In our evaluation, we only considered single-threaded applications.

Jolt does no further intervention after allowing the application to escape an infinite loop. In principle, it is possible for an application to escape an infinite loop and then crash, producing no output. In our evaluation, we inspected the source code of our applications to determine that they continue their execution without crashing, and eventually produce outputs. We anticipate that Jolt could be extended to use any of a number of program shepherding techniques to help steer programs around potential errors [28, 13, 24, 30].

8 Related Work

Researchers have previously studied the causes for program failures, including unresponsiveness, in operating systems, server applications, and web browsers [20, 17, 32, 25]. In particular, Song et al. identify infinite loops as an important cause of unresponsiveness in three commonly used server applications and in a web browser. This paper identifies the causes of eight infinite loops in existing utility applications. Our evaluation also shows that seven of these loops can be detected by checking that their state does not change across loop iterations.

(Non-) Termination Analysis: Researchers have previously suggested using program analysis to identify infinite loops during software development. Gupta et al. [14] present TNT, a non-termination checker for C programs, which identifies infinite loops by checking for the presence of recurrent state sets, which are sets of program states that cause a loop to execute infinitely. TNT uses template-based constraint satisfaction to identify sets of linear inequalities on program variables that describe recurrent state sets. Velroyen et al. [35] also propose a template-based constraint invariant satisfaction approach to identify infinite loops — though with a different invariant generation technique. Burnim et al. developed Looper, a tool that uses symbolic execution and Satisfiability Modulo Theories (SMT) solvers to infer and prove loop non-termination arguments [9].

Each of these approaches could, in principle, be used to attach to a running instance of a program and detect an infinite loop. And, in fact, developers can use Looper [9] to break into a debugging mode to prove that a suspect loop is infinitely looping. While these approaches can identify a larger class of infinite loops than Jolt — i.e., infinite loops that change state on each iteration — this power comes at the cost of symbolic execution, SAT solving, or SMT solving. Jolt, in contrast, attaches to the concrete execution of the program and uses an inexpensive detection mechanism to identify infinite loops that do not change state. In addition, Jolt provides users with the option to escape detected infinite loops and continue the execution of the program.

Researchers have also developed static analysis tools that can be used during program development to determine statically, when possible, whether each loop in the program terminates [11, 10, 8, 33]. We view these approaches as complimentary in that it would be possible to incorporate the results of static analysis into Jolt's instrumentation decisions. Namely, if it can be proven statically that a particular loop will terminate, then Jolt need not instrument that loop.

Program Repair: Nguyen and Rinard have previously deployed an infinite loop escape algorithm that is designed to eliminate infinite loops in programs that use cyclic memory allocation to eliminate memory leaks [24]. The proposed technique records the maximum number of iterations for each loop on training inputs, and uses these numbers to calculate a bound on the number of iterations that the loop executes for previously unseen inputs. To the best of our knowledge, this is the only previously proposed technique for automatically escaping infinite loops. In comparison to the approach we present in this paper, Nguyen and Rinard's technique is completely automated, but may also escape loops that would otherwise terminate.

Researchers have also investigated techniques for general program repair that could, in principle, automatically generate fixes for infinite loops [31, 27, 30, 26, 12, 36, 29]. Weimer et al. [37] have used genetic programming to automatically generate program repairs from snippets of code that already exist in the program. In their evaluation, they used their technique to generate fixes for the infinite loops in look and indent, which we also used in our evaluation. Their automatically generated fixes eliminated the infinite loops, but at the cost of some lost functionality of the application. Compared to these fixes, escaping an infinite loop enables a user to recover the complete outputs of these applications.

Handling Unresponsive Programs: Finally, we note that operating systems and browsers often contain task management features that allow users to terminate unresponsive or long-running applications or scripts. Mac OS X, for example, provides a Force Quit Applications user interface; Windows XP provides a Windows Task Manager. Web browsers also contain user interface features that alert users to long-running scripts and offer users the option of terminating these scripts [19]. However, these facilities usually offer only termination of a long-running task, while Jolt allows for the potential continued execution after the long-running loop subcomputation. Extending Jolt to work in these environments would provide the user with the additional option of detecting and escaping infinite loops in unresponsive or long-running applications.

9 Conclusion

By making applications unresponsive, infinite loops can cause users to lose work or fail to obtain desired output. We have implemented and evaluated Jolt, a system that detects and, if so instructed, escapes infinite loops. Our results show that Jolt can enable applications to transcend otherwise fatal infinite loops and continue on to produce useful output. Jolt can therefore provide a useful option for users who would otherwise simply terminate the computation.

Acknowledgements. We would like to thank Professor Armando Solar-Lezama for providing the motivating example presented in the Introduction. We would also like to thank Hank Hoffmann, Deokhwan Kim, Fan Long, Stelios Sidiroglou, Karen Zee and the anonymous reviewers for their useful comments on the earlier drafts of this paper. We would also like to thank Claire Le Goues for providing us with the bug-containing versions of look and indent benchmarks.

This research was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0905244, CCF-1036241 and IIS-0835652) and the United States Department of Energy (Grant DE-SC0005288).

References

1. Exuberant ctags. `http://ctags.sourceforge.net`.
2. GNU grep. `http://www.gnu.org/software/grep`.
3. GNU indent. `http://www.gnu.org/software/indent`.
4. GNU sed. `http://www.gnu.org/software/sed`.
5. GStreamer. `http://www.gstreamer.net`.
6. numpy. `http://numpy.scipy.org`.
7. scipy. `http://www.scipy.org`.
8. A.R. Bradley, Z. Manna, and H.B. Sipma. Termination of polynomial programs. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2005.
9. J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *International Conference on Automated Software Engineering*, 2009.
10. M. Colón and H. Sipma. Practical methods for proving program termination. In *International Conference on Computer Aided Verification*, 2002.
11. B. Cook, A. Podelski, and A. Rybalchenko. Terminator: beyond safety. In *International Conference on Computer Aided Verification*, 2006.
12. V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *International Conference on Automated Software Engineering*, 2009.
13. B. Demsky and M. Rinard. Data structure repair using goal-directed reasoning. In *International Conference on Software Engineering*, 2005.
14. A. Gupta, T.A. Henzinger, R. Majumdar, A. Rybalchenko, and R.G. Xu. Proving non-termination. In *Symposium on Principles of Programming Languages*, 2008.
15. H. Hayashizaki, P. Wu, H. Inoue, M. Serrano, and T. Nakatani. Improving the performance of trace-based systems by false loop filtering. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.
16. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization* , 2004.
17. Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? an empirical study of bug characteristics in modern open source software. In *Workshop on Architectural and System Support for Improving Software Dependability*, 2006.
18. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation*, 2005.

19. C. Metz. Mozilla girds firefox with 'hang detector', June 2010. `http://www.theregister.co.uk/2010/06/10/firefox_hang_detector/` .
20. B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of Unix utilities and services. TR #1268, Computer Sciences Department, University of Wisconsin, 1995.
21. S. Muchnick. *Advanced Compiler Design and Implementation.* 1997.
22. V. Nagarajan, D. Jeffrey, and R. Gupta. Self-recovery in server programs. In *International Symposium on Memory Management*, 2009.
23. N. Nethercote and J. Seward. Valgrind A Program Supervision Framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44–66, 2003.
24. H.H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *International Symposium on Memory management*, 2007.
25. N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.
26. J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W. Wong, Y. Zibin, M. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Symposium on Operating Systems Principles*, 2009.
27. F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Symposium on Operating Systems Principles*, 2005.
28. 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 *Symposium on Operating Systems Design and Implementation*, 2004.
29. E. Schulte, S. Forrest, and W. Weimer. Automated program repair through the evolution of assembly code. In *International Conference on Automated Software Engineering*, 2010.
30. S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. Keromytis. Assure: automatic software self-healing using rescue points. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.
31. S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference*, 2005.
32. X. Song, H. Chen, and B. Zang. Why software hangs and what can be done with it. In *International Conference on Dependable Systems and Networks*, 2010.
33. F. Spoto, F. Mesnard, and É. Payet. A termination analyzer for java bytecode based on path-length. *Transactions on Programming Languages and Systems*, 32:1–70, March 2010.
34. H. Theiling. Extracting safe and precise control flow from binaries. In *International Conference on Real-Time Systems and Applications*, 2000.
35. H. Velroyen and P. Rümmer. Non-termination checking for imperative programs. In *International Conference on Tests and Proofs*, 2008.
36. Y. Wei, Y. Pei, C. Furia, L. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *International Symposium on Software Testing and Analysis*, 2010.
37. W. Weimer, T.V. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, 2009.
38. W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Symposium on Operating Systems Design and Implementation*, 2010.