Detecting and Escaping Infinite Loops Using Bolt

by

Michael Kling

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

Masters of Engineering in Electical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2012

© Massachusetts Institute of Technology 2012. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
February 1, 2012

Certified by . . . .
Martin Rinard
Professor
Thesis Supervisor

Accepted by . . . .
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Detecting and Escaping Infinite Loops Using Bolt

by

Michael Kling

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2012, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electical Engineering and Computer Science

Abstract

In this thesis we present Bolt, a novel system for escaping infinite loops. If a user suspects that an executing program is stuck in an infinite loop, the user can use the Bolt user interface, which attaches to the running process and determines if the program is executing in an infinite loop. If that is the case, the user can direct the interface to automatically explore multiple strategies to escape the infinite loop, restore the responsiveness of the program, and recover useful output.

Bolt operates on stripped x86 and x64 binaries, analyzes both single-thread and multi-threaded programs, dynamically attaches to the program as-needed, dynamically detects the loops in a program and creates program state checkpoints to enable exploration of different escape strategies. This makes it possible for Bolt to detect and escape infinite loops in off-the-shelf software, without available source code, or overhead in standard production use.

Thesis Supervisor: Martin Rinard
Title: Professor

Acknowledgments

Thanks to Sasa Misailovic, Michael Carbin, and Martin Rinard for all of the feedback and assistance with this thesis, and all their work on the original Jolt project that this grew from. Thanks to Stelios Sidiroglou for comments suggesting the use of libunwind which ultimately proved invaluable in the implementation of Bolt. Also thanks to Deokhwan Kim and Fan Long for their feedback on earlier versions of the Jolt and Bolt papers.

Contents

1 Introduction **15**

1.1 Motivation 15

1.2 Bolt 16

1.2.1 Bolt Workflow 16

1.2.2 Experimental Results 18

1.2.3 Bolt vs. Jolt 18

1.3 Contributions 21

2 Using Bolt **25**

2.1 Example Application: PHP 25

2.2 Using the Bolt GUI 26

3 Implementation **29**

3.1 Overview of Bolt Architecture 29

3.2 Bolt Detector 31

3.2.1 Loop Structure Detection 31

3.2.2 Infinite Loop Detection 33

3.2.3 Multithreaded Loop Detection 33

3.3 Bolt Escape Tool 35

3.3.1 Unwind Escape Strategy 36

3.3.2 Break Escape Strategy 36

3.3.3 Escape Search Strategies 37

3.4 Application Checkpointing 37

3.4.1 Dynamically Loading Checkpoint Libraries 38
3.5 Bolt GUI and Python API 38
3.6 Bolt Testing Framework 39

4 Empirical Evaluation 41
4.1 Benchmark Summary 41
4.2 Methodology 44
4.3 Numerical Results 45
4.3.1 Timing Results 45
4.3.2 Loop Size Results 46
4.4 Escape Results 47
4.4.1 Continued Execution Errors 47
4.4.2 Termination Comparison 47
4.4.3 Manual Fix Comparison 48
4.4.4 Comparison of Two Escape Strategies 49
4.4.5 Jolt Comparison 50
4.5 Checkpointing Results 50

5 Detailed Case Studies 53
5.1 PHP 53
5.1.1 Infinite Loop Detection 54
5.1.2 Effects of Escaping Infinite Loop 55
5.1.3 Comparison with Termination 55
5.1.4 Comparison with Manual Fix 56
5.2 Wireshark 57
5.2.1 Infinite Loop Detection 57
5.2.2 Effects of Escaping Infinite Loop 58
5.2.3 Comparison with Termination 58
5.2.4 Comparison with Manual Fix 59
5.3 Gawk 59
5.3.1 Infinite Loop Detection 59

5.3.2 Effects of Escaping Infinite Loop 60
5.3.3 Comparison with Termination 61
5.3.4 Comparison with Manual Fix 61
5.4 Apache 62
5.4.1 Infinite Loop Detection 63
5.4.2 Effects of Escaping Infinite Loop 63
5.4.3 Comparison with Termination 64
5.4.4 Comparison with Manual Fix 64
5.5 Pluggable Authentication Modules (PAM) 65
5.5.1 Infinite Loop Detection 66
5.5.2 Effects of Escaping Infinite Loop 67
5.5.3 Comparison with Termination 68
5.5.4 Comparison with Manual Fix 68

6 Related Work **71**
6.1 Infinite Loop Detection 71
6.2 Program Repair 72
6.3 Handling Unresponsive Programs 74

7 Conclusion **75**

A Dynamic Call Stack Analysis **77**
A.1 Motivation 77
A.2 Explanation of Algorithm 78
A.3 Proof of Correctness 79

B Bolt User Manual **83**
B.1 Downloading and Installing 83
B.1.1 Requirements 83
B.1.2 Building Bolt 84
B.1.3 Running Tests 84
B.2 Running Bolt 85

B.2.1 Potential Problems 86
B.3 Developing and Extending Bolt 86
B.3.1 Adding Tests 86
B.3.2 Python API 87

List of Figures

2-1 Example PHP Program 25
2-2 Appearance of Bolt GUI After Example Use 27

3-1 Timeline for a Typical Use of Bolt 30
3-2 From Working Draft, Standard for Programming Language C++ . . 33

5-1 Script That Causes PHP to Enter Infinite Loop 54
5-2 Wireshark Loop in packet-zbee-zcl.c 57
5-3 Dissassembly of Wireshark Infinite Loop 57
5-4 Awk Program and Input That Cause Infinite Loop 60
5-5 Placed in httpd.conf to Trigger an Infinite Loop 62
5-6 Entries in .pam_environment File That Trigger an Infinite Loop . . . 66
5-7 Code in pam_env.c That Causes the Infinite Loop 67

A-1 Simple Demonstration Program 78

List of Tables

3.1 Summary of Bolt Tests 40

4.1 Studied Infinite Loops 42

4.2 Infinite Loop Time Statistics 46

4.3 Infinite Loop Memory and Length Statistics 47

4.4 Summary of Escape Results 48

4.5 Comparison With Jolt 51

4.6 Results of BLCR Checkpointing 52

A.1 Call Stacks During Execution 79

Chapter 1

Introduction

1.1 Motivation

Infinite loops can cause programs to become unresponsive, preventing them from producing useful output, causing users to lose important data, and causing systems to unproductively consume power. Infinite loops that occur in server applications can lead to denial of service or other attacks, which can cause widespread downtime, and loss of services for users.

While infinite loops may be a problem in practice, attempting to detect infinite loops may be met by objections that the Halting Problem is unsolvable. In 1936 Alan Turing proved that a general purpose algorithm to determine for all pairs of programs and inputs if the program will terminate, cannot exist. The key part of this statement is *all* input programs.

When programs do not terminate, they may be executing an infinite loop. The Halting Problem does not preclude a program from detecting some subset of non-terminating programs, for example, infinite loops that repeat state. In this case, state refers to all the values in memory that a program could read or write from to influence execution. Modern computers execute deterministically, and contain only finite memory. As a result, any program can only exist in a finite number of possible states, and so must eventually either terminate, or return to an earlier state. Other operations, such as network I/O, could also effect whether a program terminates,

but these are addressed separately. Through the benchmarks presented here, we demonstrate that in practice, programs which have entered an infinite loop tend to return to an earlier state quickly, and this can be detected effectively.

The detection of infinite loops presents only part of the problem. While it may be informative for a developer to know when a program is executing in an infinite loop, attempting to correct the execution of the program, and simultaneously trying to prevent any greater damage occuring to the system, presents another important hurdle.

1.2 Bolt

We present a new system, Bolt, for escaping infinite loops in stripped x86 and x64 binaries. When an application becomes unresponsive, a user can attach Bolt to the application. Bolt then examines the execution to determine if the application is in an infinite loop. If so, Bolt tries multiple loop escape strategies to find a strategy that enables the application to continue successfully. If the application successfully escapes from the loop, Bolt detaches from the application.

1.2.1 Bolt Workflow

Bolt consists of three components: a detector module, an escape module, and a checkpoint module. Bolt starts by using the detector module to monitor the execution of the application and detect repeated program states as follows. When Bolt attaches to an application, it records the current instruction. Whenever execution returns back to that instruction, Bolt takes a snapshot of the current state and compares that snapshot to the last N-1 snapshots it has taken (here N is a configurable Bolt parameter). If one of the previous snapshots matches, Bolt has detected an infinite loop. The loop consists of all instructions that executed between the last and current execution of the original instruction.

The escape module next explores different loop exit strategies in an attempt to find an escape strategy that allows the program to continue successfully. First, Bolt

finds the stack frame that contains the loop (Bolt is designed to detect infinite loops that may cross procedure boundaries). As it monitors the application, Bolt tracks stack frames as they are pushed onto and popped off of the call stack. The stack frame that contains the loop is the lowest frame that is always on the stack during the execution of the loop. This is referred to as the loop's 'base stack frame'.

Once the escape module has this stack frame, it can explore alternatives from two escape strategies:

- **Jump Beyond Loop (Break):** The escape module unwinds the stack to the loop's base stack frame. It next finds the set of instructions that are 1) in the loop and 2) in the loop's base stack frame. It then finds the instruction immediately after this set of instructions and jumps to that instruction.

- **Return From Enclosing Procedures (Unwind):** The escape module executes a return to one of the stack frames above the loop's base stack frame. This immediately escapes the loop by exiting the procedure that contains the loop. Each of the stack frames above the stack frame that contains the loop gives Bolt another loop escape alternative. Additionally, Bolt may modify function return values before continuing execution. The return values can be used to indicate the success or failure of particular functions.

There is, of course, no guarantee that any specific strategy will enable the application to continue to execute successfully. Bolt therefore gives the user the opportunity to try different alternatives — if the user does not like how the application responds to one alternative, Bolt can roll back to the checkpoint and try another alternative.

Applicability

Bolt operates on unmodified binaries, has no effect on the execution unless it is attached, dynamically identifies the boundaries of loops, and dynamically generates loop escape alternatives. Bolt works with both single and multithreaded applications. This makes it possible for Bolt to detect and escape infinite loops in off-the-shelf

software without available source code, the need to recompile the application to insert instrumentation, or overhead in standard production use.

1.2.2 Experimental Results

We applied Bolt to thirteen infinite loops in ten applications. Bolt detects twelve of the thirteen infinite loops. Even though Bolt does not determine that the thirteenth loop is actually an infinite loop, it does allow the user to escape the loop and continue to execute successfully.

For seven of the loops, Bolt enables the application to provide the same correct output as a subsequent version of the application with the infinite loop error eliminated via a developer fix. For the remaining six loops, Bolt enables the application to provide some but not all of the correct output (see Chapter 4 for details). For five of these six loops, we believe that users would find this partial output more useful than the output from simply terminating the application.

1.2.3 Bolt vs. Jolt

The closest related work to Bolt is Jolt [16], a compiler-based tool for detecting and escaping infinite loops. Bolt differs from Jolt as follows:

- **Stripped Binaries:** Bolt works by attaching directly to executing x86 or x64 binaries, then monitoring the execution to detect and escape infinite loops. Jolt, in contrast, relies on a compiler to insert the instrumentation required to detect and escape infinite loops. Bolt's approach has the following advantages:

  - **No Required Source:** Because Bolt operates directly on (potentially stripped) binaries, there is no need for source code and no need to recompile with a specialized compiler. Users can apply Bolt directly to any application that will execute on their platform regardless of how the application was obtained or whether any part of the application is available except the executable code. Jolt, in contrast, is restricted to applications

with available source code that the Jolt compiler can successfully compile for their platform.

  – **Full Coverage:** Because Bolt works with binaries, it can detect and escape infinite loops that happen to appear in any part of the application, including loops that appear in external components such as libraries. Because application build environments typically link in external components without recompilation, Jolt typically does not instrument these external components and is therefore unable to detect and escape infinite loops within such components.

  – **On Demand Usage Model:** Users can download and attach Bolt to a running application after it becomes unresponsive. There is no need to recompile the application anticipating that it may enter an infinite loop.

  – **No Overhead in Production Use:** Because Bolt affects the execution of the application only when it is attached, the application executes with no overhead whatsoever during normal production use. The inserted Jolt instrumentation, in contrast, imposes between 2% and 10% overhead whenever the application executes.

- **Loop Structure Detection:** Bolt detects loops by dynamically monitoring the application as it executes to detect repeated states. Bolt's ability to detect loops is therefore largely independent of how the loop is expressed in the source code. Jolt, in contrast, relies on static program analysis to find loops. Bolt can detect loops that Jolt can not and this enhanced capability is important in practice — several benchmarks, including PHP and Gawk have loops that Bolt can detect but Jolt cannot.

- **Exploration of Multiple Escape Alternatives:** Bolt implements several approaches for escaping the loop — in particular the break and unwind strategies. Moreover, the unwind strategy is not limited to return back to the immediate caller — it can return back to any caller currently on the call stack with different return values.

Before it tries the first escape alternative, Bolt can checkpoint the state of the application. If one attempt to successfully escape the loop does not work, Bolt can roll back to the checkpointed state and try the next alternative. This capability is important in practice as different escape strategies work best for different applications.

Jolt, in contrast, implements only a single escape strategy (jump to the instruction following the loop) and provides no capability to explore different strategies.

- **Support for Multithreaded Applications:** Bolt supports multithreaded applications — it monitors each thread separately for infinite loops and tracks potential interactions between threads. Specifically, Bolt monitors for operations the C++ standard specifies must eventually occur in all threads, including synchronization and atomic operations, I/O calls, and shared memory access, further addressed in Section 3.2.3. Jolt, in contrast, works only with single threaded applications. Because many modern applications use multiple threads, Bolt's ability to handle multithreaded applications is important in practice. For example, Bolt is able to detect and escape an infinite loop in the Wireshark multithreaded application (which is beyond the reach of Jolt).

- **Infinite Loops Across Multiple Iterations:** Jolt detects infinite loops by checking if successive iterations produce the same state. Bolt, in contrast, can detect infinite loops across N successive iterations (where N is a configurable Bolt parameter). So, for example, Bolt can detect infinite loops in which the state repeats only every other or every third iteration. This enhanced capability is important in practice — the state of the the infinite loop in PHP repeats only every four iterations, while the state of the infinite loop in gawk repeats after up to 20 iterations.

Because of these advantages, I believe that Bolt's binary-based approach delivers a more usable, efficient, and comprehensive infinite loop detection and escape tool than Jolt's compiler-based approach. I also note that Bolt currently implements the

core systems building blocks required to address a much larger range of anomalies and errors. Specifically, Bolt's monitoring, checkpointing, and search capabilities make it possible to respond to an execution error by trying different recovery strategies until one enables the application to continue successfully [32].

1.3 Contributions

This thesis makes the following contributions:

- **Bolt:** Users can download and install the Bolt system, an inclusive collection of tools designed to detect and escape from infinite loops in x86 and x64 binaries, using checkpoints to explore multiple loop escape strategies.
- **Implementation**
  - **Bolt Detector:** I implemented a unique algorithm in the detection tool for dynamically reconstructing the call stack during analysis. In addition, the detector takes and compares up to N snapshots on loop iterations. Multithreaded detection is supported; when one or multiple threads are executing in an infinite loop, the detection will be run in parallel. Multi-threaded loop identification reflects the semantics in the C/C++ standards on the actions of executing threads.
  - **Bolt Escape Tool:** The escape tool is entirely separate from the detection tool, though it makes use of the analysis results. The escape tool monitors the target program after escape, to judge if escape was successful. I've implemented two escape strategies, the unwind strategy which can be adjusted using several parameters, and the break strategy.
  - **Checkpointing:** Application checkpointing is implemented using a third party library, BLCR. Dynamic loading of the necessary checkpoint libraries allows the ability to checkpoint a program to be added on demand.
  - **Bolt GUI/Python API:** I've implemented a GUI to combine all of the functionality in one accessible way, and add a Search feature allowing a

user to easily try multiple escape strategies on a target program. A user can use the Python API to interact with the escape and detection features of Bolt programmatically.

  - **Bolt Testing:** I've written 17 tests to evaluate different aspects of the detection and escape mechanisms. The testing architecture allows the test suite to be easily extended.

- **Results**

  - **Benchmarks:** I've done experiments on a total of 10 benchmark applications and 13 infinite loops. Five applications were new for the Bolt tool, and include PHP, Wireshark, Gawk, Apache, and Pluggable Authentication Modules (PAM).
  - **Case Studies:** I performed a detailed analysis of all five new case studies. This analysis describes each infinite loop, the detection, results of escape, and comparison to alternatives including termination and later fixes by developers.
  - **Timing Results:** Infinite loop detection and escape are both efficient. Detection takes between 0.0003 seconds and 5.85 seconds. Escape takes a maximum 0.024 seconds. When no tools are operating, they present no overhead for the target program.
  - **Loop Size Results:** I evaluated each benchmark for the size of and memory use in the infinite loop. Memory footprints in each loop are a maximum of 1 kb. Total instructions in the loops vary from 1 to 74k, with a median of 150.
  - **Escape Results:** I evaluated the continued execution of each benchmark after using each escape strategy. All benchmarks have at least one escape strategy which does not result in any execution errors. Compared to simply terminating the program, 11/13 benchmarks can use one escape strategy to give a better result. Compared to later developer fixes of these bugs,

7/13 loops have a Bolt fix that is equivalent and the remaining 6/13 all give at least a partial result. Out of these 13 benchmarks, the original 8 benchmarks from Jolt were also tested to compare the success of Bolt vs. Jolt. For all of them, at least one escape strategy gives a result equivalent to Jolt.

– **Checkpointing Results:** I evaluated the effectiveness of checkpointing. Taking a checkpoint was successful in 9/11 applications (some benchmarks are from the same applications), with a median time of 0.177 seconds. Reasons for unsuccessful checkpoints are discussed in Section 3.4. The median checkpoint size was 408 kb.

Chapter 2

Using Bolt

This section discusses the usage of Bolt from a user perspective, presenting a sample application, and steps a user should take to detect and escape from an infinite loop, resulting in the continued execution of the application.

2.1 Example Application: PHP

Consider a user who has written a long running script in PHP. This script executes for a period of time performing some computation, and outputs the result at the conclusion. The script might appear as shown in Figure 2-1.

```
<?php
$result = 0
// ...
// Executing lots of computation
// ...

$result = $result + 2.2250738585072011e-308;
printf( "Final result: %.17e \n", $result);
?>
```

Figure 2-1: Example PHP Program

Unfortunately for the user, if he or she is using version 5.3.4 of PHP, when reaching the second to last line of this program containing the double literal value added to `result`, the script will enter an infinite loop. From the user's perspective, the

program can not reach the last line, the final result is not printed, and thus the script is essentially useless.

A technically inclined user might be able to attach to the running script using a debugger, and inspect register and memory values to identify the result without needing to restart the script, or move the instruction pointer past the infinite loop. The Bolt framework attempts to provide functionality to an average user that will allow them to attempt repair of the executing program, so it will reach the last line, and make the desired result visible.

Before using Bolt, a user will need to download and install the tool, covered in Appendix B.

2.2 Using the Bolt GUI

To start using Bolt, the user starts the Bolt UI, shown in Figure 2-2. The window presents a list of running processes, sorted by CPU usage. In this case, the php script in an infinite loop would be visible at the top of the list. In the figure shown, Bolt has already successfully escaped from an infinite loop so PHP is no longer visible in the process list.

At this point, the user begins interacting with the GUI in a series of steps, in an attempt to fix the program.

- First, the user selects the target process and chooses Detect. The result of this detection is logged in the UI. Detection can either be successful, or not. The GUI will also show potential warnings to indicate that an infinite loop was detected, but not with complete certainty. Additional details are stored in Bolt log files.

- Before attempting any escape strategies, a user may wish to take an application checkpoint. This will allow a user to restore the program to that checkpoint, and try other escape strategies if the first doesn't give a useful result. The 'Checkpoint' button will take an application checkpoint. In this case we assumed

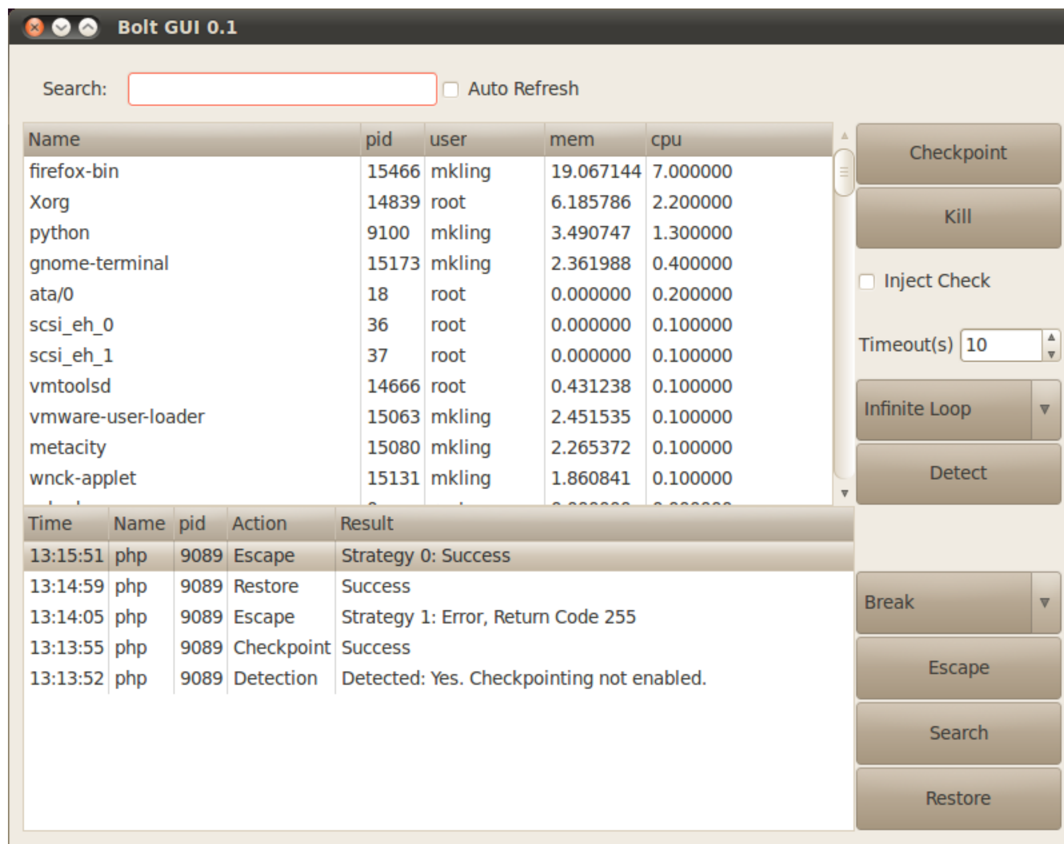

Figure 2-2: Appearance of Bolt GUI After Example Use

the application already had checkpointing enabled, however, a user can attempt to enable it dynamically by selecting the 'Inject check' button before running detection.

- The user chooses an escape strategy from a drop down menu, selects the target process, and hits Escape. The result is then logged to the UI, with additional information found in the logs directory.

- In this case, the first strategy was unsuccessful, and for the particular example of PHP, the user would see an error, `Parse error: syntax error, unexpected $undefined in test.php`, and not get their desired output. As a result, the user can use the Restore button to restore the program from the checkpoint.

- After restoring, a user can attempt a different escape strategy. In this case, this strategy is successful, and the user is able to view the output of the script.

- The Search button automates the process of trying escape strategies and restoring from checkpoints. After trying one escape strategy, Bolt evaluates if it was successful or not. If not, the program is restored from a checkpoint, and the next escape strategy is executed. This process repeats until a successful escape strategy is found, or until all have been exhausted.

Chapter 3

Implementation

In this chapter we present the implementation details of Bolt's components.

3.1 Overview of Bolt Architecture

The Bolt system contains a detection component, escape component, and a checkpoint component. These components can be used separately, but are tied together with the Bolt User Interface. A typical user workflow might proceed as shown in Figure 3-1. This reflects the workflow described in Chapter 2, with more technical details.

- The program enters an infinite loop and becomes unresponsive. The user launches Bolt.
- The Bolt detector attaches to the program, and begins monitoring to determine 1) if it is in a loop, and, if so, 2) whether or not the loop is definitely an infinite loop. Once completed, the detector saves the result to a file, and detaches from the program.
- Before attempting any repair strategies, a checkpoint may be taken of the program. This checkpoint contains all the state of the process.
- The Bolt escape tool attaches to the program. Depending on the parameters passed to the escape tool, a specific escape strategy will execute, and the tool

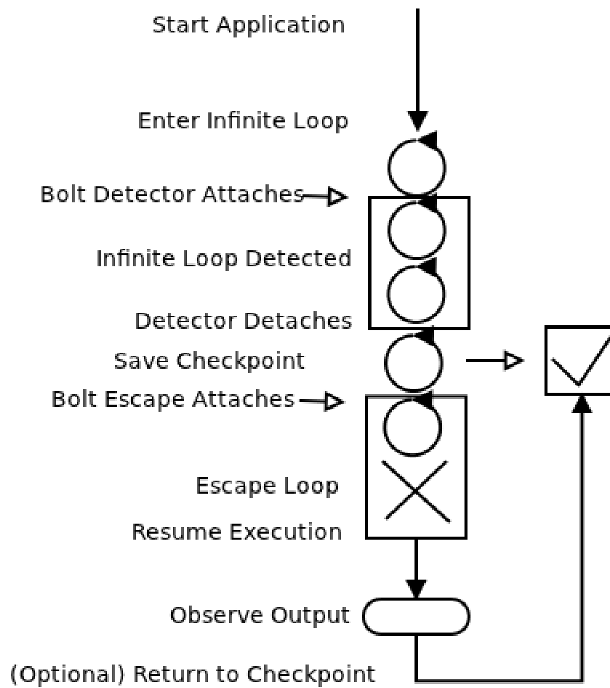

Figure 3-1: Timeline for a Typical Use of Bolt

will attempt to monitor the program to see if it completes successfully.

- Depending on the scenario, the program may be automatically restored to the earlier checkpoint. In this case, the Escape tool may again attach to the program, with different parameters.

Each of the Bolt modules is implemented using the following tools.

- **Bolt Detector:** The Detector attaches to the target application using the dynamic instrumentation framework Pin [26]. The Pin framework can analyze x86, x64, and IA-64 binaries, on Linux or Windows. Pin is used to monitor memory accesses and function calls.

- **Bolt Escape Tool:** The Escape tool uses `ptrace` system calls to attach to the target application. These are used to set breakpoints, and execute the escape strategies. As implemented it is compatible with x86 and x64 binaries on Linux.

- **Checkpoint Module:** The checkpoint module is implemented using BLCR (Berkeley Lab Checkpoint/Restart [2]). This checkpoint library is implemented

as a Linux kernel module, and stores details of the program state.

- **Bolt User Interface:** The user interface is implemented in Python, and uses the `psutil`[9] library (included in the distribution) to provide information on running processes, and PyGTK[10] (generally available in Python distributions) for the GUI.

3.2 Bolt Detector

The Bolt Detector includes several components. The first is for detecting loop structure in the binary. This component carefully monitors the state of the call stack, so that loops which cross function boundaries can be properly detected. The second component takes memory snapshots, and uses them to detect when state is not changing between loop iterations. The third component monitors actions taken by the program that could affect proper detection of loops in multithreaded programs, including shared memory access, synchronization, and atomic operations.

3.2.1 Loop Structure Detection

Since Bolt does not have access to the source code of the target application, it must determine the structure of loops in the application dynamically.

The Bolt Detector attaches to the application using Pin. After initially attaching, the detector saves the value of the instruction pointer. Each time the detector returns to this original instruction pointer, Bolt must determine if it has completed an iteration of the loop. The instruction could be in any function called from the loop. Merely seeing the same instruction pointer does not indicate that an iteration of the loop has completed. Thus, Bolt must be sure we have reached the same location in the program, from the same sequence of function calls. Because a program may not have call stack information (or a consistent stack pointer register), Bolt dynamically reconstructs the call stack of functions within the infinite loop.

Bolt uses Pin to monitor all calls and returns from functions. As Pin allows the

loop to execute one instruction at a time, we dynamically reconstruct two partial call stacks. The first, S_c, represents the call stack at the current instruction. The second, S_i, represents the call stack at the original instruction pointer. These call stacks are compared when returning to the orginal instruction pointer, to see if the it has been reached with the same sequence of function calls. These call stacks contain function return addresses. For each instruction, the following rules are evaluated in order to modify these call stacks.

1. If the previous instruction was a function return, check if S_c is empty. If so, insert the current instruction address at the bottom of stack S_i, as if S_i was a queue.
2. If the previous instruction was a function return, and S_c is not empty, pop the top element off the S_c stack.
3. If the current instruction has the same address as the initial instruction, compare S_c and S_i. If they are the same, one loop iteration has completed.
4. If the current instruction is a function call, push the return address of this function onto S_c.

The stack S_c is modified in the same way as the actual call stack, with the exception that elements are not popped off if the stack is already empty. Thus, we can be sure that S_c is identical to the top of the actual call stack. In the case that S_c is empty after a function returns, as in rule 1, we are reaching a stack frame we have never seen. In this case, we can be certain the original instruction was called from this stack frame, and thus we add this address to the bottom of the stack S_i.

Loop structures implemented using traditional loop constructs can be identified, as well as arbitrary loops implemented using goto constructs in the same function. We note that some loop structures generated using a mix of setjmp/longjmp and recursive function calls would not be detected as loops. A more complete analysis including a proof of correctness for this algorithm is included in Appendix A.

3.2.2 Infinite Loop Detection

At the end of each iteration of the loop, Bolt takes a snapshot of the program's memory and register state. Bolt saves a fixed (but parameterized) number of snapshots that enables it to determine if the loop is infinite. Specifically, for any sequence of loop iterations and corresponding snapshots, if execution returns to the same state, then the loop is infinite. Bolt takes snapshots only of values modified in the previous loop iteration. When comparing snapshots, the set of memory and registers contained in the snapshots are compared first, followed by their contents.

Library Routine Abstraction: Some routines may modify internal state, that is not visible to the program. An example would be a write call that modified the position of a file cursor. If the position value is never used, this state would never affect loop termination. Bolt uses an analogous strategy to [16], and allows the specification of routines whose side effects are ignored.

3.2.3 Multithreaded Loop Detection

```
The implementation may assume that any thread will eventually do
one of the following:
  -- terminate,
  -- make a call to a library I/O function,
  -- access or modify a volatile object, or
  -- perform a synchronization operation or an atomic operation
[Note: This is intended to allow compiler transformations such as
removal of empty loops, even when termination cannot be proven.
--end note]
```

Figure 3-2: From Working Draft, Standard for Programming Language C++

Users are likely to encounter multithreaded applications when using programs that contain a GUI or more complex features. Bolt detects infinite loops in multithreaded programs on a per-thread basis. When Bolt attaches to a multithreaded application, it begins tracking the memory accesses of each thread separately. When a given thread completes one iteration of a loop, snapshots of memory accessed only by that

thread are compared. State information that is shared by multiple threads will appear in multiple thread snapshots.

In addition to memory snapshots, Bolt tracks other actions taken by each thread. This implementation is based on the C++ standard. The most recent draft of the C++ standard [11] contains a quote, in Figure 3-2, that details exactly what a compiler can expect a valid C++ program to do.

In particular, if a thread does not eventually terminate, make a call to a library I/O function, etc., its behavior is undefined. The Bolt detector monitors each thread for these types of operations, as follows:

- **Library I/O:** When available, function names are compared against a list of known library I/O functions.
- **Synchronization/Atomic Operations:** Pin provides functionality to monitor these operations.
- **Volatile/Shared Memory:** The addresses of memory accesses in each thread are monitored separately. Once Bolt believes an infinite loop has been detected, Bolt searches for intersections between these sets to identify shared memory.

Once Bolt detects identical snapshots for a given thread, execution of all threads is halted using a global lock. If any of the above tests indicate calls to library I/O, atomic operations, etc., the user is warned that the thread may not be in an infinite loop.

In general, the Bolt detector can rarely be certain a particular thread in a multithreaded program is in an infinite loop. Other threads could eventually take some action to influence the execution of a thread suspected to be in an infinite loop. However, by modelling the implementation off of the C++ standard specification, Bolt indicates to the user when an infinite loop has likely been detected, and warns of any potential issues.

In some cases, such as single threaded programs in loops that do not change state and contain no I/O operations, Bolt can be certain the program is in an infinite loop.

This does not prevent other changes in execution, such as another program sending a signal, or an OS interrupt.

3.3 Bolt Escape Tool

We evaluate two choices of escape strategies: *unwind* (which forces the execution to continue from the return address of the function containing the loop) and *break* (which forces the execution to continue at the first instruction after all instructions executed by the loop).

To force the program to escape from the loop using one of these strategies, Bolt executes the Bolt escape tool. This tool is implemented using `ptrace` system calls, and is currently compatible with Linux x86 and x64 applications. The tool proceeds as follows:

1. Before exiting, the Bolt Detector writes the results of analysis to a file, then detaches all dynamic instrumentation. These results include the application's process pid (or the thread id for the particular thread in an infinite loop), a breakpoint address located at an instruction in the function containing the infinite loop, and the expected stack pointer value when reaching this instruction.

   The Bolt escape tool is invoked separately, and this information is passed in on the command line. It optionally takes an address that execution should continue at after escaping from the loop. If omitted, the unwind escape strategy is used. In this case, the escape tool can take an optional parameter specifying what value to return from the target function when unwinding the stack.

2. The escape tool then attaches to the application with `ptrace`, places the breakpoint, and allows the application to continue executing until it hits this breakpoint. This point is identified by the Bolt Detector as part of the loop structure reconstruction. The Bolt Detector always chooses this point by looking at all instructions executed in the loop's base stack frame, and choosing the instruction with the maximum address.

3. When the application hits the breakpoint, the escape tool compares the stack pointer of the application with the expected stack pointer that was passed as an argument. This ensures the breakpoint has been hit in the correct stack frame. If not, execution continues until the breakpoint is hit in the correct stack frame.

4. Once the breakpoint is hit, one of the two escape strategies is executed. Once completed, the Bolt Escape tool detaches from the application, and allows it to continue uninstrumented without any overhead.

3.3.1 Unwind Escape Strategy

For the unwind escape strategy, the libunwind library [28] is used to unwind one or more frames of the stack, effectively forcing the program to return from the current function and continue execution from the return address. The return value can be optionally modified, by placing specific values in the return registers. This enables the unwind strategy to also consider returning common error codes from the function, such as `NULL` or -1 (similar to the approach used by [35]). The libunwind library uses a combination of exception handling information and stack walking to perform call stack reconstruction.

In some cases the program binary may not contain enough information for libunwind to accurately identify a stack frame to unwind to. In these cases, Bolt reports to the user that it is unable to perform escape and does not modify the application's execution.

3.3.2 Break Escape Strategy

If the break escape strategy is being used, `ptrace` modifies the instruction pointer to the destination value, and execution continues. This destination value is generated earlier, by the Bolt detector. The destination address is always the next instruction after the breakpoint, in the loop's base stack frame.

After the Bolt Escape tool has executed its escape strategy, it waits for the application to terminate so it can report if the target application terminates successfully.

When the program continues executing without terminating, the escape tool will be terminated externally from the Bolt UI after a timeout has occurred.

Once the Bolt Detector and Escape tool have completed or been terminated, the application runs without instrumentation, and thus no overhead.

3.3.3 Escape Search Strategies

The search function that appears in the Bolt GUI provides an automated method to explore multiple escape strategies on a target program. In order to use the search function, a user must run detection on the program and take a successful checkpoint. The search function will then execute a loop. On each iteration, one escape strategy will be attempted, and the result recorded. If the target program does not terminate after a timeout, the user attempts to determine if the escape strategy was successful by observing the target program. After executing one strategy, the program will be terminated (if necessary), then restored from the checkpoint.

The end result of using the Search function is a summary of using each possible escape strategy on a target program. An escape strategy is deemed successful if the target program returns zero after executing an escape strategy, or if the strategy times out, and the user indicates success. Non-zero return values or program termination by a signal are considered unsuccessful.

3.4 Application Checkpointing

While the original Jolt tool allowed only for forward execution after escaping from a loop, Bolt allows the exploration of several escape strategies through the use of checkpointing.

Checkpointing in the Bolt framework is done using BLCR (Berkeley Lab Checkpoint/Restart, [2]), which is implemented as a kernel module. This checkpoint saves memory and register state, file system state, and works for single and multithreaded applications. Some resources, including sockets and other resources, cannot be checkpointed by BLCR.

In order to take a checkpoint of an application, the target program must either have been started using the BLCR tools, which force the program to load a shared library at runtime, or have loaded the library in some manner after execution began. Since BLCR is used as a drop in library, improved support implemented by their developers would be immediately supported by Bolt, or the entire library could be replaced by another checkpointing system. Direct support in the Linux kernel, which would entirely eliminate the need for a third party library, is ongoing development.

3.4.1 Dynamically Loading Checkpoint Libraries

Using Pin, we can force applications that were not started using the BLCR tools to load the necessary libraries to support checkpointing at runtime. In particular, we used Pin to inject calls to the `dlopen()` function as a part of the detector tool, before any analysis occurs. This is an optional part of the Bolt detector. When using Bolt with this feature, the system can be invoked on demand without the need to start a target program using specific tools. This approach reflects the approach used in Process Hijacking [20], to enable checkpointing in running programs.

3.5 Bolt GUI and Python API

The Bolt GUI is designed to make the use of the various Bolt components as straightforward as possible. The interface is divided into two main panes, showing the list of running processes, and a log of actions taken by Bolt tools and their results. The remainder of the interface is a collection of elements that expose the functionality of each of the remaining Bolt modules.

Most of the Python functions used to implement the GUI act as wrappers, which gather the correct arguments for and make calls to execute the Bolt detector, escape tool, or checkpointing commands. Some of these make calls to the Bolt Python API. This API exposes two main functions, discussed further in Appendix B.

- `detect(pid, ...)`: Given a process ID, the Bolt detector attaches and at-

tempts to detect an infinite loop. Results are logged to a file, and return values indicate if a loop was detected and possible warnings.

- `escape(pid, ...)` Given a process ID, the Bolt escape tool reads the results from an earlier call to `detect`, and attempts to escape from an infinite loop. Additional arguments can specify what escape strategy to use. Return values indicate if the escape was successful, or if a timeout occurred.

3.6 Bolt Testing Framework

Testing for the Bolt framework is implemented in a Python script. This script uses the Bolt Python API to attempt to detect and escape from infinite loops in 9 simple test applications. Each test application has been implemented to test some aspect of the Bolt system, and they attempt to showcase a variety of types of infinite loops. The test suite is easily extensible simply by adding more tests to a file.

Each test is specified using the following criteria:

1. Name of the test
2. If the Bolt Detector tool should detect an infinite loop
3. Which escape strategy to use (break/unwind)
4. If using unwind strategy, what return value to use
5. The expected return value from the program after escaping

The success of each test is determined by comparing the results of the Bolt Detector with the expected result, and the return value of the target program after using the Bolt Escape tool with the value expected. A summary of the tests implemented and their purpose is shown in Table 3.1.

Test Name	**Purpose**
simple	Simplest possible infinite loop.
simple2	Infinite loop in nested function.
simple3	Requires call stack analysis to identify loop iteration.
multistate	Loop iterates through several repeating states.
strat	Different escape strategies result in different program return values.
retval	The program returns whatever value used by the unwind strategy.
breakerr	The break strategy causes a memory error.
multithread	Two threads are spawned, one runs an infinite loop that can be detected.
notdetected	An infinite loop which can't be detected, but still escaped from.

Table 3.1: Summary of Bolt Tests

Chapter 4

Empirical Evaluation

In this chapter, we present a description of the benchmarks used to evaluate Bolt, our experimental methodology, a summary of escape and numerical results, and comparison with the Jolt tool.

We evaluated Bolt by applying it to detect and escape thirteen infinite loops in ten benchmark applications. Bolt was able to detect twelve of the thirteen infinite loops (the remaining changes state on every iteration).

We have been able to identify infinite loops in all the benchmarks that existed for the Jolt tool, and have investigated other, highly visible bugs in applications like Wireshark, PHP, and Apache. Forcing programs to escape from the infinite loops has in all cases resulted in either a partial or complete result, as compared to results from the correction later made by the developers. In the case of an interactive GUI application such as Wireshark, the program returns from a state of accepting no inputs, and appearing completely frozen, to a state of normal operation, allowing a user to perform additional analysis or exit the program normally.

4.1 Benchmark Summary

Table 4.1 presents the version numbers of the benchmarks that contained the infinite loop (Version) and the later version with the infinite loop eliminated via a developer fix (Reference Version).

Benchmark	**Version**	**Reference Version**
php	5.3.4	5.3.5
wireshark	1.4.0	1.4.1
gawk	3.1.1	3.1.2
httpd	2.2.18	2.2.19
pam	1.1.2	1.1.3
indent	1.9.1	2.2.10
ctags-fortran	5.5	5.5.1
ctags-python	5.7b (646)	5.7b (668)
grep-color	2.5	2.5.1
grep-color-case	2.5	2.5.1
grep-match	2.5	2.5.1
ping	20100214	20101006
look	1.1 (svr 4)	-

Table 4.1: Studied Infinite Loops

In total we evaluated thirteen infinite loops in ten applications. We chose the loops based on our ability to reproduce infinite loop errors reported in error reports. The applications range from common utilities like grep to large, multithreaded GUI applications like Wireshark.

- **php:** PHP is a widely used general purpose scripting language, commonly used for web development and in server side applications [4]. This infinite loop occurs when the PHP interpreter attempts to parse a certain floating point value from a string. It occurs only in 32-bit builds of PHP, but presents a denial of service (DoS) risk to server applications if they receive this floating point value [8].
- **wireshark:** Wireshark is a network protocol analyzer, allowing a user to capture and analyze traffic on a computer network [5]. This infinite loop can occur when parsing certain ZigBee ZCL packets that contain malformed attributes. This infinite loop can be triggered by opening a previously recorded file of packet information, or by a remote user sending corrupt packets [21].
- **gawk:** Gawk is the GNU implementation of the AWK pattern scanning and text processing language [3]. This infinite loop occurs when attempting to parse an input containing nested single and double quotes. It is a result of a flaw in

the regular expression implementation in gawk [6].

- **httpd:** The Apache HTTP server is designed to be an efficient and secure server that is extensible and provides HTTP services [1]. An infinite loop occurs in the Apache Runtime Library in a function used for matching URL regular expressions [7].
- **pam:** Pluggable Authentication Modules, or PAM, are used to provide authentication mechanisms for Linux programs. This infinite loop is triggered when a certain PAM module parses a file containing environment variables, and overflows due to an evironment variable being longer than an internal PAM buffer [19].

In addition to these five benchmark applications, we also used the eight benchmark applications in the Jolt benchmark suite:

- **ctags:** We investigated two infinite loops:
  - **ctags-fortran:** The fortran module in version 5.5 had an infinite loop when parsing certain declarations separated by semicolons.
  - **ctags-python:** The python module in version 5.7 had an infinite loop that occurs when parsing certain multi-line strings.
- **grep:** We investigated three infinite loops in grep version 2.5. These loops occur as a result of a zero length match:
  - **grep-color:** Occurs when grep is configured to display the matching part of each line in color.
  - **grep-color-case:** Occurs when grep is configured with case insensitive matching, and to display matching parts of each line in color.
  - **grep-match:** Occurs when when grep is configured only to print the matching part of each line.
- **ping:** An infinite loop occurs when parsing certain fields of reply messages.
- **look:** A dictionary lookup program. An infinite loop occurs as a result of a dictionary entry not terminated by a newline character.

- **indent:** An infinite loop occurs parsing a final line containing a comment and no newline character.

4.2 Methodology

For each of the benchmarks, we performed the following:

- **Reproduction:** Using bug reports for each application, we found appropriate inputs that caused each infinite loop.

- **Detection:** For each benchmark, we ran the application and allowed it to enter an infinite loop. We then started Bolt, and determined if it was able to identify an infinite loop.

- **Escape:** For each benchmark, after using Bolt to attempt detection of each infinite loop, we chose to force the program to escape from the loop. We then observed the continued execution of each program. We observed manually to see if the resulting output was well formed, and we used Valgrind to determine if there were any invalid memory accesses or leaks.

  To do this, we ran each application under Valgrind and caused the application to enter the infinite loop. Once in the infinite loop, we used the Valgrind facility allowing a program to break into GDB. Once in GDB, we manually altered the program to emulate each of the escape strategies used by Bolt, quit from GDB, and allowed the program to continue running under Valgrind.

  This manual escape is necessary as an application running under Valgrind cannot currently be properly attached using Bolt. While not technically impossible, an application running under Valgrind cannot be attached using Pin while executing, and so the application would need to run under Bolt from the start. This would require a reworking of the Bolt detector, which expects to attach to an application already in an infinite loop.

We note that all infinite loops in our benchmarks can be triggered deterministically and repeated for any program run.

- **Comparison with Termination:** We compared the output obtained from terminating the application with the output from applying each of the Bolt escape strategies.

- **Comparison with Manual Fix:** Using later versions of the applications where the infinite loop has been manually corrected, we compared the outputs of these later application versions with the results of different Bolt escape strategies.

- **Comparison between Strategies:** Our evaluation considers two possible escape strategies for use by Bolt, the Unwind and Break strategies. We compared the output seen from using each of these strategies, to see if some benchmarks produce more useful output using one escape strategy than another.

- **Comparison with Jolt:** For each of the original benchmarks evaluated by the Jolt tool, we compare the results of escape using Jolt and using Bolt.

- **Checkpointing:** For each benchmark, we took a checkpoint of the application before using Bolt. We determined for each application whether it was possible to take a checkpoint, and if restoring execution from the checkpoint after using Bolt resulted in any differences or lost data.

4.3 Numerical Results

4.3.1 Timing Results

We looked at the overhead of using the Bolt tool, in particular, the amount of time taken to detect an infinite loop after the detector attaches to the application, and the amount of time taken to escape the loop and continue executing. The data are summarized in Table 4.2. These times were obtained by taking the median of five instances of attaching to the program with Bolt while in an infinite loop.

Benchmark	Detection Time (s)	Escape Time (s)
php	1.34759	.008469
wireshark	0.00031	.006355
gawk	0.59836	.004242
httpd	0.16784	.005140
pam	5.85644	.010560
ctags-fortran	0.12222	.004327
ctags-python	0.14132	.002224
grep-match	2.45763	.012965
grep-color	2.14553	.010507
grep-color-case	1.34774	.006526
ping	0.02571	.002649
look	0.16406	.002235
indent	–	.023956

Table 4.2: Infinite Loop Time Statistics

4.3.2 Loop Size Results

In addition to gathering the time required to detect and escape from each infinite loop, we also gathered information on the footprint, or number of bytes in the program state that Bolt recorded in each snapshot, and the length of each infinite loop, represented by the number of instructions (of any type) that are executed in each loop iteration. The data are summarized in Table 4.3. The footprints were divided into footprint size of registers and memory. In some cases, programs may not return to the same state until executing multiple loop iterations, for example the gawk benchmark. It is possible that different loop iterations will have a different sized footprint or have a different length. In these cases, we have taken the maximum of all iterations.

We computed the correlation between the lengths of each loop in instructions, and the time to detect each loop in seconds, and found an R squared value of 0.78. This positive correlation suggests loops that contain more instructions will take longer to detect, as the time taken for Bolt to execute the loops is greater. Initial setup overhead is minimal, as shown by the detection time for the smallest loop in Wireshark.

Benchmark	Reg. Size (b)	Mem. Size (b)	Length
php	128	688	7784
wireshark	192	0	1
gawk	192	169	390
httpd	192	8	106
pam	192	0	74606
ctags-python	192	24	78
ctags-fortran	192	0	239
grep-match	192	1001	1575
grep-color	192	1001	1577
grep-color-case	192	1093	1740
ping	192	0	21
look	192	108	153

Table 4.3: Infinite Loop Memory and Length Statistics

4.4 Escape Results

This section discusses the results of the continued execution of each benchmark after using each escape strategy.

4.4.1 Continued Execution Errors

We observed the continued execution when using each of the two escape strategies on these infinite loops and recorded any invalid memory references. These data are summarized in columns two and three of Table 4.4. Those entries marked with an asterisk (*) indicate resulting execution errors that were handled automatically by error handling code within the application. For example, the memory errors observed in gawk are handled by an internal signal handler.

In the benchmarks tested, bad escape strategies quickly resulted in fatal errors such as invalid memory accesses, which the escape tool reported with its return value. Successful escape strategies quickly completed execution, which the escape tool also reported.

4.4.2 Termination Comparison

We also compared results with termination of the program, and observed if escaping from the loop gave additional useful output. We did this comparison for both strate-

Benchmark	Errors After:		vs. Termination		vs. Manual Fix		Best
	Unwind	**Break**	**Unwind**	**Break**	**Unwind**	**Break**	**Strategy**
php	Parse*	None	Better	Better	None	Same	Break
wireshark	None	Parse*	Better	Better	Same	Partial	Unwind
gawk	None	Memory*	Better	Same	Partial	None	Unwind
httpd	None	None	Better	Better	Same	Partial	Unwind
pam	None	None	Better	Same	Same	None	Unwind
ctags-fortran	None	Memory	Better	Same	Partial	None	Unwind
ctags-python	None	None	Better	Better	Partial	Partial	Same
grep-match	None	None	Same	Same	Partial	Partial	Same
grep-color	None	None	Better	Better	Partial	Partial	Break
grep-color-case	None	None	Better	Better	Partial	Partial	Break
ping	None	None	Better	Better	Same	Partial*	Unwind
look	None	None	Same	Same	Same	Same	Same
indent	None	None	Better	Better	Partial	Same*	Break

Table 4.4: Summary of Escape Results

gies of escaping from the loop. In 11 out of 13 benchmarks, using Bolt provided more output than terminating the program.

In the remaining two cases, no further output was expected after entering the infinite loop, so termination and using Bolt gives the same result. The results are summarized in columns four and five in Table 4.4. Those entries marked as 'Better' produced more output than resulted from terminating the program, and those entries marked 'Same' produced no more output than terminating.

4.4.3 Manual Fix Comparison

Using later versions of each program in which the infinite loops have been corrected by developers, we compared the results of escaping from the loop using one of the two escape strategies, with the results of execution using the fixed program version. The results are summarized in columns six and seven of Table 4.4. Those entries marked as 'Same' produced an identical output as that seen in the fixed program version. Those marked as 'Partial' produced some useful output, but not the complete result seen from the fixed program version. Those marked as 'None' produced none of the output the manual fix produces after the loop, possibly as the result of an error.

For all of the infinite loops, at least one escape strategy resulted in a partial result.

In seven of the thirteen infinite loops, one of the escape strategies provided output identical to that of the manually corrected application. In the cases marked by an asterisk (*), the output after escaping from the loop contained the entire expected output from the fixed version, but also executed additional code that gave more output. The result is arguably acceptable, as this extra output could be ignored by the user. In the case of indent, the result is arguably semantically identical, as the output is a C text file, and the result of escape using the Break escape strategy differs by only 1 additional character of whitespace from that of the corrected application. In the case of ping, the extra output results in displaying time stamp information, parsed from a corrupt packet. The corrected version of the program ignores this part of the packet.

In the case of PHP, the results appear inconsistent with the results compared to termination, where the unwind strategy produced better results than termination, but produced none of the results of the manual fix. Discussed more in Section 5.1, using the unwind strategy gives a parsing error, which provides more information than termination by indicating the line of code the error occurs in. However, this does not represent any of the output produced using the fixed version of PHP, and so compared to the manual fix, 'None' is the appropriate result for the unwind strategy.

4.4.4 Comparison of Two Escape Strategies

Lastly, for each of the infinite loops, we evaluated which of the two escape strategies was more effective. In cases where both provided no output, or both provided identical output, the two were judged to be the same. In cases where one provided more output, including cases where both provided a partial result as compared to the later manual fix, but one provided more output, that escape strategy was judged more effective. Results are summarized in column eight of Table 4.4. In four of the thirteen infinite loops, the break strategy gave the best output, and in six cases, the unwind strategy gave the best output.

Note that for the strategy that produced the more useful output, the code was manually inspected to ensure no memory leaks occurred after executing this escape

strategy.

4.4.5 Jolt Comparison

For the original six benchmark applications that were analyzed by Jolt, we compared the results of using Bolt to the original published results of escape using the Jolt tool. This comparison uses both of the Bolt escape strategies. In all cases, at least one of the Bolt escape strategies was able to provide results identical to the Jolt escape strategy. The results are summarized in Table 4.5. The remaining five benchmarks are out of the scope of Jolt: Wireshark is multithreaded, PHP and Gawk execute multiple states before repeating, and the Apache and PAM loops occur in library code.

Those entries marked 'Same' produced the same, or nearly identical output as Jolt, and those marked 'Worse' produced errors or less output than the original Jolt escape. In the case of grep, the unwind strategy produces worse results in two cases, as the rest of a line is not printed after a match is found. In the case of indent, the output produced by the break escape strategy differs by one character of whitespace, which does not affect the semantics of the C source file. We also compared the loop detection times, which for Jolt were less than one second. Some benchmarks are detected faster by Bolt, and some take more time. Bolt has more overhead per instruction, as every instruction has some instrumentation. However, Jolt has greater up front overhead, as it must load the loop structure information from the binary initially. In the worst case, for the grep-match benchmark, Bolt was five times slower at detection, but was still in detected in less than three seconds. In the best case, for ping, Bolt was eleven times faster at detection.

4.5 Checkpointing Results

To test the success of application checkpointing on each of the 10 benchmark applications, we started each application with the inputs that triggered an infinite loop. After the application entered the loop, we used the Bolt checkpoint interface to take

Benchmark	Unwind	Break
ctags-fortran	Same	Worse
ctags-python	Same	Same
grep-match	Same	Same
grep-color	Worse	Same
grep-color-case	Worse	Same
ping	Same	Worse
look	Same	Same
indent	Worse	Same

Table 4.5: Comparison With Jolt

a checkpoint of the application. We then terminated the running application and attempted to restore it. For 8 out of the 10 applications, this was successful. Wireshark and Apache were not successful, as the checkpointing system of BLCR does not support sockets, and certain types of files, such as `/dev/urandom`. Successful support for sockets would require a multi-process checkpointing scheme that BLCR does not currently support. Since BLCR is used as a drop in library, any further development for this support could be immediately used, or the entire library could be replaced by another checkpointing system. The results are summarized in Table 4.6.

These results include the amount of time taken to checkpoint each application. Each application was run five times with the inputs causing an infinite loop. After entering the infinite loop, the time to take a checkpoint was measured. The median of the five detection times is shown. In the case of ctags, two different versions of the program were used, since different versions of the program contained different infinite loops. In the case of grep, we tested only version 2.5. For all applications except for indent, the variance between checkpoint times was minimal. The indent infinite loop continuously allocates memory. Thus, taking a checkpoint later in the execution of this program must record more program state, and will take a longer time. The maximum time any of the indent checkpoints took during testing was 1.53 seconds.

The checkpoint results also show the size of the checkpoint file created. BLCR stores the entire checkpoint in one file in the same directory as the executable file. The contents are describe in Section 3.4.

Benchmark	**Success**	**Time**	**Size (kb)**
ctags-5.5	Yes	0.166	360
ctags-5.7	Yes	0.136	448
grep	Yes	0.147	408
ping	Yes	0.177	360
look	Yes	0.181	212
wireshark	No	–	–
php	Yes	0.221	2192
gawk	Yes	0.139	484
indent	Yes	0.736	228
httpd	No	–	–
pam	Yes	0.269	943

Table 4.6: Results of BLCR Checkpointing

Chapter 5

Detailed Case Studies

This chapter provides a detailed analysis of the results of using Bolt on five of the benchmark applications. These applications range from important client applications and libraries, including Wireshark, Gawk, and the PAM libraries, to equally important server side applications, including Apache and PHP.

5.1 PHP

PHP version 5.3.4 for 32-bit x86 processors when compiled with the gcc compiler[1] contained an exploitable bug in the parsing of certain strings to floating point values [8]. The error occurs in the loop starting from line 2313 in the `zend_strtod` function (in file `zend_strtod.c`). This function is used during the conversion of strings (that appear either as input or constants in PHP code) to floating point values. This function takes as input a pointer to the target string and returns a double value to which the part of the input string is converted and a pointer to the end of parsed input string.

The infinite loop within this function consists of 254 lines of code, with complex control flow structure, including calls to the helper functions and reading and writing to a large number of local variables. [2]

Figure 5-1 shows an example program that triggers the infinite loop. For an input

[1] PHP's official binary distribution is compiled with gcc.

[2] We omit the code of the infinite loop due to its length. External references, such as [33] provide a detailed description of the loop, the causes of the error, and the applied correction.

string representing a specific number – the value of the variable `$d` in the example program, the loop starts executing infinitely. When using a vulnerable version of PHP, the parser enters an infinite loop on line 2 of the user's program, when attempting to read the constant value.

The loop iterates until it produces a double value which is the closest to the input string literal. The error arises when the loop continuously tries to make an adjustment to reach its desired double value, but the computation does not make progress due to the interplay between the gcc compiler and the CPU's floating point unit. Namely, the computation of intermediate results inside the FPU is performed in 80-bit extended precision registers, but the final result is converted to the regular 64-bit IEEE-double value. However, for the value of variable `$d` the rounding causes the adjustment variable to lose precision after conversion. This results in adjustment not affecting the value of the result and missing the exit condition of the loop.

```
1:  <?php
2:  $d = 2.2250738585072011e-308;
3:  printf( " %.17e \n", $d);
4:  ?>
```

Figure 5-1: Script That Causes PHP to Enter Infinite Loop

5.1.1 Infinite Loop Detection

This loop is complex, with each iteration of the infinite loop accessing over 20 local variables and executing several levels of nested function calls. This loop repeats the same execution pattern after 4 iterations. None of these iterations affect the value of the result (the double value) and its adjustment, and thus does not affect the loop's exit condition.

A closer examination of the changing state reveals that at each iteration of the loop the computation allocates and frees temporary heap data structures, which represent unbounded integers. The state in the loop that changes in every iteration are the pointer values to these data structures. For these data structures PHP uses a custom memory manager, bounded in size. After 4 loop iterations, the memory manager

starts returning the same sequence of (previously allocated and deallocated) memory locations, which Bolt identifies as identical to the previously seen state. Note that Jolt does not detect this loop (as it looks only for the identical state in two consecutive loop iterations), but Bolt is able to detect it since it extends the lookup to multiple loop iterations.

5.1.2 Effects of Escaping Infinite Loop

When using the Unwind escape strategy, PHP terminates and gives the error, pointing to the user's source code: `Parse error: syntax error, unexpected $undefined on line 2`. When the unwind library attempts to restore the earlier stack frame and continue execution, it correctly changes the instruction and frame pointers, but other registers remain unchanged, resulting in a wrong value passed in the floating point registers. This causes an error in the parser and PHP returns a parsing error message.

When using the Break escape strategy, PHP continues to execute from an instruction inside the loop that is in a branch not taken by the original computation, but which changes the values of the adjustment variables. This change does not change any of the significant digits of the resulting double value, but allows the computation to exit the loop during the next iteration, at the same location and with the same result as the manually fixed version of the program. The user's script is properly parsed and executed, printing out a double value, with no errors visible to the user.

For the Break escape strategy, we used Valgrind to check for latent memory errors, using the approach described in Section 4.2. Valgrind does not report any memory errors, and the inspection of the program's code shows that the branch at which Bolt continues the execution ensures that the currently allocated memory is properly freed.

5.1.3 Comparison with Termination

If the infinite loop is triggered during the PHP compilation phase, as in the example above, termination results in no PHP code being executed. Since PHP performs parsing before executing any part of code, it will not execute even the code that is

located in the script before this particular double literal appears.

Although the Unwind escape strategy does not execute the code, it still provides a parsing error on the line where the value appears in user's code. The Break strategy does enable the program to terminate and to execute every instruction of the code, including the instruction which contains the double value `$d`.

The infinite loop could also be triggered when reading the double value from some input source, such as a file or an HTTP request. In this case, termination will result in no code past this point being executed. The Unwind escape strategy will terminate the script (but will log the location of the error). The Break strategy will continue the script execution past this point.

We note here that PHP has a facility for detection and termination of long running scripts which is invoked by setting a limit on the maximum execution time. However, it cannot recognize and terminate the script that contains this infinite loop. PHP checks for the elapsed time of the script only after executing (interpreting) complete PHP instructions. But, this infinite loop appears in the middle of interpreting a single PHP instruction, so PHP will not be able to detect that the script is executing above the limit.

5.1.4 Comparison with Manual Fix

The developers manually fixed the application by making local variables that represent adjustment volatile, which forces the compiler to store these values in memory after each write (thus implicitly running the conversion from 80-bit to 64-bit values), which results avoids the undesired rounding of these variables.

We compared the result of executing the program from Figure 5-1 on faulty version of PHP on which we apply Break escape strategy and on the manually corrected version 5.3.5 of PHP. In both cases, both printed results had all digits identical.

5.2 Wireshark

Wireshark is software commonly used for network traffic monitoring, recording, and analysis. It provides a graphical interface which lets user analyze the network traffic in the real-time. Version 1.4.1 of Wireshark contains an infinite loop in the ZigBee wireless protocol module [21]. The infinite loop can be triggered by reading a malformed packet file, or by a remote user maliciously sending corrupt packets.

```
1: while ( *offset < tvb_len && i < ZBEE_ZCL_NUM_ATTR_ETT ){
2:  if( tree )
3:  {
4:    // ...
5:    i++;
6:    // ...
7:  }
8: }
```

Figure 5-2: Wireshark Loop in packet-zbee-zcl.c

Figure 5-2 presents a simplified version of the loop. The variable `tree` is a pointer. If `tree` is `NULL`, then no code inside the loop is ever executed, and the condition for ending the loop will never be satisfied. This loop is located in the function `dissect_zcl_discover_attr_resp()`, starting from the line 1192 in file `packet-zbee-zcl.c`.

5.2.1 Infinite Loop Detection

When compiled with gcc -O2, the generated code checks the value of the `tree` pointer. If it is not `NULL`, a conditional branch continues execution of the instructions inside the loop. If it is `NULL`, then execution continues in an infinite loop that lasts for one instruction, shown in Figure 5-3. We note that the loop has such a simple structure due to compiler's loop unswitching optimization (which splits the original loop into the two loops representing individual branches, and moves the `if (tree)` condition outside to control which loop to execute).

```
0x7f66ccc51136: jmp  0x7f66ccc51136
```

Figure 5-3: Dissassembly of Wireshark Infinite Loop

This makes the infinite loop in Wireshark the shortest of all our benchmarks.

However, detection does present some challenges. In particular, Wireshark is a multithreaded application, and during the detection process, multiple threads may be executing. However, in the case of this benchmark, the infinite loop makes no memory accesses, and we can be certain that another thread changing shared memory will never cause this loop to exit. Note that unlike Bolt, Jolt would not be able to recognize this loop as it does not distinguish memory accesses between different threads: Jolt's snapshots will contain memory accesses from different threads, which will interfere with the ability to detect an infinite loop if *any* of the threads are changing the program's state.

5.2.2 Effects of Escaping Infinite Loop

The application becomes responsive after Bolt escapes the infinite loop both with Unwind and Break strategies. The application also presents the network traffic that caused the infinite loop and all subsequent traffic. Since there is no additional code after the loop, Unwind behaves exactly like breaking from the loop.

The Break strategy transfers the execution of the program to the code within the if branch (since it is located below the infinite loop in the program's binary), which attempts to parse a corrupted packet. However, every operation within the body of if branch checks again whether the tree is equal to `NULL` (in which case they stop processing the packet). The executing code returns a status message denoting that the packet is malformed.

5.2.3 Comparison with Termination

Terminating the program could result in lost data, if a user had been logging packets and was unable to save, as well as a potential loss of security in a case where Wireshark is used to monitor for potentially malicious network traffic. In contrast, both of Bolt's escape strategies help application become responsive again and continue analyzing traffic, including the packet that caused the infinite loop.

5.2.4 Comparison with Manual Fix

The manual fix by the developers simply moved the check for `NULL` values of the `tree` pointer outside the loop, so no code inside the loop is ever executed. Since the function has no code after the loop, the result of the manual fix is equivalent to the Unwind escape strategy. The output of the manual fix is identical to that of the Unwind escape strategy. When using the Break escape strategy, the program presents all results that the manually fixed program produced, but also outputs the additional `''[Malformed Packet]''` status message.

5.3 Gawk

Gawk is the GNU implementation of the awk language. Awk is a text processing language commonly used for manipulating files and data. Version 3.1.1 of gawk contained an infinite loop in its regular expression library [6].

Figure 5-4 shows an awk program, and a corresponding input file, that will cause this infinite loop, when executed on the command line using the command, `gawk -f loop.awk input`. The awk program contains a regular expression designed to match pairs of double quotes (written as two single quote characters), and surround them with `<em>` tags. The result of this substitution on each line is then printed out. In this case, the loop is triggered when gawk encounters a string with nested double quotes.

5.3.1 Infinite Loop Detection

The infinite loop in Gawk is very complex. It occurs in the file `regex.c`, beginning on line 5615. This loop exits when a complete match is found, or if a match fails. In total this loop contains 1996 lines of code. The developer's response to this bug report was:

This is a bug who-knows-where in the guts of the regex.[ch] library. Unfortunately, I treat that as a black box, and have no idea how to fix it.[3]

[3]http://osdir.com/ml/gnu.utils.bugs/2002-10/msg00046.html

loop.awk:

```
{sub(/''(.?[^']+)*''/, "<em>&</em>"); print}
```

input:

```
''Nested '' apostrophes''
''No nested apostrophes''
```

Figure 5-4: Awk Program and Input That Cause Infinite Loop

Each iteration of this loop changes state, and consecutive iterations through the loop follow different execution paths. Depending on where in this large loop the Bolt detector attaches, it will take a different number of iterations to detect a repeated state. The number of different states varied between 10 and 20 in our testing.

5.3.2 Effects of Escaping Infinite Loop

The result of escaping the loop using the Unwind strategy depends on the value that Unwind causes the program to return (Unwind can return the typical error values such as 0/`NULL` or -1; recall Section 3.3). As a reminder, Bolt creates a checkpoint before trying any escape strategy, and if the strategy fails due to, e.g., program crash, it can restore the program to the starting state and try another escape strategy. This allows Bolt to explore different return values.

If Unwind forces the return value 0 or uses the value residing in the eax register from the function with the infinite loop, the program terminates with a memory error. Gawk contains a signal handler designed to handle this type of error, and prints the error message, `gawk: loop.awk:1: (FILENAME=input FNR=1) fatal err: internal error`. At this point Bolt can restore the state of the program and try an alternative escape strategy.

If Unwind forces the return value -1 from the function with the infinite loop, the program will continue the execution, and will produce the following output for the example input file:

```
''Nested <em>'' apostrophes''</em>
<em>''No nested apostrophes''</em>
```

We can see that a match is found on the first line, and a replacement made, though it is not the first match on the line. The match on the next line is then processed without any problem. Running with valgrind detected no memory leaks when using this escape strategy. A manual inspection of the regex code used shows that -1 is used as an error code representing no match found in the regex library.

After escaping from the infinite loop with Break strategy, gawk terminates the execution as the result of the memory error, printing the same error message as seen previously for some cases of the Unwind strategy.

5.3.3 Comparison with Termination

Terminating the program halts gawk from processing any further data in input files. This could cause a user to receive a partial result when using gawk to process a large input. Our strategy allows gawk to process more input, while only affecting the result on lines that caused the infinite loop.

5.3.4 Comparison with Manual Fix

The developers fixed this infinite loop as a part of a complete rewrite of awk's pattern matching engine. In the bug report response, the developer announced the new version of the pattern matching library, which at the time was in development.

We used an updated version of gawk, 3.1.2, to process this input. The result of processing this input file was:

```
<em>''Nested ''</em> apostrophes''
<em>''No nested apostrophes''</em>
```

In general, regular expressions match from left to right. Based on this rule, the fixed version of gawk provides a more accurate result than the unwind escape strategy, which finds the second match on this line. The remaining lines of the output are identical in both case.

5.4 Apache

Apache version 2.2.18, in the included Apache Portable Runtime Library, version 1.4.4, contains an infinite loop bug in code designed for string matching [7]. This error occurs in the `apr_fnmatch` function, in apr_fnmatch.c, loop starting on line 199 and continuing until line 362. This function takes a pattern to match, an input string, and a set of flags that controls properties of the matching. The function returns an integer, containing the value `APR_FNM_NOMATCH=1` (no match), or 0, representing a match. The bug is a result of an attempt to fix a previous bug, which used recursion and in some cases could cause high cpu usage. The entire function `apr_fnmatch` was rewritten when this bug was introduced.

```
<Location "/*/WEB-INF">
deny from all
</Location>
```

Figure 5-5: Placed in httpd.conf to Trigger an Infinite Loop

Figure 5-5 shows an entry in httpd.conf, which will cause an infinite loop when a client makes certain requests to the server. This entry is intended to prevent any connections from accessing directories matching the regular expression `"/*/WEB-INF"`. WEB-INF directories are usually used to hold configuration files and Java class files for JSPs (Java Server Pages). When starting Apache, and using a browser to navigate to a page on the server, Apache will test if the address matches the target location. If so, the connection will be denied. This check occurs in server/request.c, in the `ap_location_walk` function, for every web request that is received.

In the vulnerable case, if `apr_fnmatch` is called with a pattern of `"/*/WEB-INF"`, and a string of `"/test"`, after entering the infinite loop, the string will be completely consumed, and point to `'\0'`, a null string, and the pattern will be partially consumed, pointing to `"/WEB-INF"`. There are no checks in the loop to see if string has been completely consumed, and thus, the loop continues forever in an attempt to match the rest of the pattern.

5.4.1 Infinite Loop Detection

Bolt is able to detect this infinite loop quickly, as no state is changing on each iteration, and only string library calls are made within the loop when it reaches a fixed state. This infinite loop occurs in the Apache Runtime Library code, loaded at runtime. As a result of runtime linking, depending on system settings, the addresses of the loop may change on each execution, which presents no problem for the Bolt tool. The original Jolt tool does not consider code beyond that stored in the executable.

The original Jolt tool would require that this library have been originally compiled with the Jolt compiler in order to run detection.

5.4.2 Effects of Escaping Infinite Loop

On the server side, using the break strategy will allow the server to escape from the loop, and continue execution. The `apr_fnmatch` function will return 0, representing a match, as the code checks simply to see if the input string (not the input pattern) has been completely consumed, and returns success. As a result, on the client side, the user will see a 403 Forbidden error. When accessing from a browser, the browser will often generate a second request for the icon file favicon.ico, which will trigger the infinite loop a second time on the server side. This icon is used as a status bar icon for this page, and is requested by all modern graphical browsers. Using the same escape strategy allows the server to escape again from the loop.

When using the unwind strategy with a return value of 0, representing a match, the result is a 403 forbidden error for the client (the same as using the break strategy). The server will continue executing outside the infinite loop. Using the unwind strategy with any other value will be interpreted as no match, and the server will continue to process the request. Internally, Apache makes recursive calls to the function `ap_process_request_internal`, and for the same request, the infinite loop will be hit again, in a different stack frame. We can continue to use the same escape strategy of unwinding using a non zero return value, and Apache will continue to process the request and generate a response. Generally our escape tool checks the value of the

stack pointer before running an escape strategy to ensure it is escaping from the loop in the correct stack frame, but it can be optionally disabled. This is ideal for cases like this where the same loop is repeatedly hit in different stack frames.

Making a manual inspection of the infinite loop, we can see that no memory is allocated or freed during the loop execution, and the loop consists entirely of pointer arithmetic and comparisons. None of our escape strategies introduce memory leaks or other errors.

5.4.3 Comparison with Termination

When running Apache in most configurations, several processes run to handle requests simultaneously, to help distribute load. Terminating one of these processes would spread the load to the other processes, and the Apache control process may automatically start another process.

On the client side, the request will be hanging, and terminating the target process will result in the server terminating the connection without sending any reply data, which will be reported by the browser. The break strategy gives a partial response to the client, in particular a permission error, instead of terminating the connection. Arguably, this could provide more information, that a client could then give to an adminstrator to track down an error. The unwind strategy for non zero return values gives the client the intended response, namely, the content of the desired page, without a permissions error.

5.4.4 Comparison with Manual Fix

The developers fix, in APR version 1.4.5, added a check at the start of the loop body to see if the input string has been completely consumed, and a check after the loop body that returns success when both the input pattern and input string have been completely consumed, representing a match. With this fix, instead of entering an infinite loop, the above inputs would break from the loop once the input string consumes every character, then return 1 (no match).

This fix results in an identical result to our unwind escape strategy with return value of 1. Requests are handled in an identical manner. We note that this benchmark presents some security issues. For this particular case, using Bolt escape strategies at no point allows a user to access data that matches the given location regular expression. However, it is possible to imagine a case where using Bolt could alter the execution of the target program in a way that would allow a remote user to access data that was intended to be blocked. This is a risk inherent in altering a program's execution at runtime. In this case, we envision an administrator testing the results of different escape strategies locally, and determining which fix provides the best results before using it on production servers.

5.5 Pluggable Authentication Modules (PAM)

Pluggable Authentication Modules or PAM implements authentication mechanisms as shared libraries, used by many Linux programs and utilities, including login, su, sshd, etc. PAM version 1.1.2 contained an infinite loop that could be triggered by an overflow in environment variable expansion routine [19]. This routine, `_expand_arg`, is called in the `pam_env` module, in pam_env.c. The loop begins on line 553. As input, the routine it takes a pointer to a `pam_handle` structure (used in this function only for logging information) and a pointer to a string that contains the variable to be expanded.

As PAM modules are not standalone, but used from other programs, we will use as an example a user calling `su - $USER`. This call will prompt for the user's password, which will be authenticated using PAM. By default, PAM will then read from the file `~/.pam_environment`, and variables set in this file will be part of the user's execution environment after the call to su starts a new shell. The `pam_env` module is used to read this file, and calls the `_expand_arg` function. This function contains a loop which scans the input string for variables to expand.

In Figure 5-6 we show an input file for `~/.pam_environment` that will trigger an overflow condition and the infinite loop. The `pam_env` module contains a constant

```
EVIL_FILLER_255 DEFAULT=BBBBBBBB... [repeated 255 times]
EVIL_FILLER_256 DEFAULT=${EVIL_FILLER_255}B
EVIL_FILLER_1024 DEFAULT=${EVIL_FILLER_256}${EVIL_FILLER_256}\
                         ${EVIL_FILLER_256}${EVIL_FILLER_256}
EVIL_FILLER_8191 DEFAULT=${EVIL_FILLER_1024}${EVIL_FILLER_1024}\
                         ${EVIL_FILLER_1024}${EVIL_FILLER_1024}\
                         ${EVIL_FILLER_1024}${EVIL_FILLER_1024}\
                         ${EVIL_FILLER_1024}${EVIL_FILLER_256}\
                         ${EVIL_FILLER_256}${EVIL_FILLER_256}\
                         ${EVIL_FILLER_255}
EVIL_OVERFLOW_DOS DEFAULT=${EVIL_FILLER_8191}AAAA
EOM
```

Figure 5-6: Entries in .pam_environment File That Trigger an Infinite Loop

`MAX_ENV` set to 8192. Note this is a limitation of PAM, not of Linux, though environment variables that are too large will prevent `execve()` calls from occurring, as there is a limit on the combined argument and environment size. Each time a variable is expanded, the resulting total length of the string is compared against this constant. If the string is too long, this is logged using the `pam_syslog` function. When expanding the `EVIL_OVERFLOW_DOS` variable shown, this check occurs, shown in Figure 5-7, and the overflow condition is logged. However, once this is logged, no local variables are changed, and the loop continues executing, looping forever, and filling the system log with repeat messages (the code comments themselves question if this logging is a good idea).

5.5.1 Infinite Loop Detection

Bolt is able to detect this infinite loop. The loop iterates through a very large number of states before repeating, not because of local changes in the `_expand_arg` function, but because of buffer changes made by the logging function `pam_syslog`, which eventually calls libc functions that contain internal buffers. If we enable library routine abstractions in a similar manner to the grep benchmark, as described in the Jolt paper [16], and ignore all state changes in the libc methods called by the `pam_syslog` function, the loop is still detected, but in a shorter time, and indicates only 1 state.

```
while(*orig) { /* while there is still some input to deal with */
// ...
  if((strlen(tmp) + 1) < MAX_ENV) {
    tmp[strlen(tmp)] = *orig++;
  } else {
    /* is it really a good idea to try to log this? */
    D(("Variable buffer overflow: <%s> + <%s>", tmp, tmpptr));
    pam_syslog(pamh, LOG_ERR, "Variable buffer overflow: \
                              <%s> + <%s>", tmp, tmpptr);
    // return PAM_BUF_ERR; // Patch later introduced
  }
// ...
}
```

Figure 5-7: Code in pam_env.c That Causes the Infinite Loop

5.5.2 Effects of Escaping Infinite Loop

Using the break strategy does not result in escaping from the infinite loop. Instead, execution continues inside the loop along a different path, previously unseen by the Bolt detector, for one iteration. One character 'A' is interpreted as an unrecognized escape character. On subsequent loop iterations, execution returns to the original path and remains in an infinite loop.

Using the unwind strategy with non-zero values allows the loop to escape. The unwind return value is compared with the success value of zero, and an error is reported up the call stack. The loop is then entered a second time, in this case due to the nature of PAM modules. Different PAM modules are loaded in a stack, and more than one may execute different parts of PAM. Escaping from the loop a second time, in our original example use of su, will result in an error message, and termination of su: `su: Critical error - immediate abort`.

Using the unwind strategy with a value of zero also allows the loop to escape, and continue executing. The return value of zero indicates the call has been successful. For the same reason as above, the loop is entered a second time, but can be escaped again using a return value of zero. Tracing the continued execution in the example case of 'su', the program makes a call to `fork()`, and the child process calls `execve()` to start a new shell. The environment in this shell contains all the

variables declared in the `.pam_environment` file, including `EVIL_OVERFLOW_DOS`, which has the value `"${EVIL_FILLER_8191"`, as the expansion routine never had a chance to copy the expanded variable into this string.

Unfortunately, using the escape tool of Bolt to execute this escape strategy causes the shell process that starts to execute in the background. The shell expects to run in the foreground, and attempts to read directly from the terminal. As a result, the process repeatedly receives the SIGTTIN signal, which in the case of bash, is ignored. This effectively puts the shell in an infinite loop (which we were able to successfully detect using Bolt), and leaves it unusable. Using gdb instead to manually execute this escape strategy does allow the shell to execute in the foreground and a user to interact with it as expected.

By manual inspection, we can see that no memory is allocated in this function, until the very end, when the expanded variable is copied to the original buffer passed as input. The unwind strategy never executes this code, and all other local variables are found on the stack, so no memory leaks occur.

5.5.3 Comparison with Termination

Terminating the process in the example case of using 'su' has the same effective result as using the successful unwind escape strategy. However, a huge variety of programs use the PAM libraries, and each may handle an internal PAM error in a different way. In many cases, a fatal error in the authentication code would result in an application choosing to terminate, but this may give it additional time to log information and clean up resources before terminating. Using the break strategy leaves a user no worse off than before. If desired, they can attempt to use other escape strategies at this point.

5.5.4 Comparison with Manual Fix

We compared with the patched version of the PAM libraries, on ubuntu released as 1.1.2-ubuntu8.4. Using su, the result was identical to what we saw using the non zero

unwind strategy. Inspection of the code shows that the return value of `_expand_arg` is handled internally and will always result in the same termination condition when returning small non zero values. Since this function is declared static, it will not be visible externally and using this escape strategy will always give the same result as the developer fix for programs that use the `pam_env` module.

Arguably, our result when using an escape strategy with value 0 presents an alternative fix for this code. Rather than returning an error message, which is reported as a general PAM error to the program using PAM, the offending environment variable could be truncated, or discarded, and PAM could log a warning message. As currently implemented, PAM does not provide any information to an application about the details of the error, and details of the overflow are available only in the system logs.

As in the Apache benchmark, we also analyze it from a security perspective. In this case, the infinite loop occurs only after the user has been authenticated. In addition, Bolt, and in general Pin and the ptrace system call can not be used to attach to processes that a user does not have access permissions for. Thus, a malicious user is prevented by the operating system from using Bolt in an attempt to maliciously attach to and alter the execution of a root process. We also note that this infinite loop is unique in that it may disable an entire system. We discovered this during testing when we were temporarily unable to login to a machine equipped with a vulnerable version of the PAM library. We created an environment file on the machine to trigger the overflow. Unfortunately, the user login process uses PAM, and after a system restart, the machine would hang after correctly entering the password. The hope would be that a user invokes Bolt and identifies and fixes a problem before the entire system becomes unusable.

Chapter 6

Related Work

6.1 Infinite Loop Detection

Researchers have developed several techniques for detection of infinite loops, that use symbolic program execution [23, 13]. These approaches are able to detect a wider set of infinite loops, but they incur additional overhead in their use of symbolic execution and SAT or SMT solvers. Unlike Bolt, these tools do not attempt to continue execution of the program by exiting the loop and require source code. As a related problem, researchers have also developed techniques for program debugging and verification that ensure that a program does not contain any infinite loops [18, 17, 12, 37]. In principle, Bolt detection mechanism could be extended to use arbitrarily complex mechanisms to identify non responsive programs (without affecting the escape mechanism).

The Jolt tool [16] was the first to show that in practice, infinite loops can often be detected simply by seeing if the state of the program changes between iterations. However, Jolt compared only the two most recent snapshots. In the case of our PHP benchmark, Jolt would not be able to detect this infinite loop, as several iterations must complete before PHP returns to an identical state.

In addition, Jolt made the first attempt to escape from infinite loops, and evaluate the results of continuing execution. However, Jolt used only one escape strategy, similar to the break strategy of Bolt, but with the destination to break provided by the

program source code. Bolt, while inspired by the work done in Jolt, does not require the program source code, and does not introduce instrumentation that adds overhead to the application. In addition, Bolt provides an extensible environment for escape strategies, which easily allows for more sophisticated development of program repair strategies in the future. Bolt takes care of other practical concerns not addressed by Jolt. These include working with multithreaded applications, and the use of process checkpointing to allow Bolt to make multiple repair attempts to programs. The result is that Bolt provides a tool that is usable in practice.

6.2 Program Repair

In addition to Jolt [16], researchers have previously investigated a number of other techniques for general failure recovery and general 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 [29]. 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.

In Jim Gray's paper on computer system failures [22], he discusses the phenomenon of soft failures in software. In particular, he proposes the technique of reinitializing software to an earlier state and retrying failed operations. Several checkpointing techniques are described to provide fault tolerance, including backup processes for a primary process, and process checkpointing provided by the kernel.

The Recovery-Oriented Computing and Microrebooting [30, 14, 15] approaches propose that software systems can be designed to enable recovery from failures by rebooting a component (or rolling it back to a known good checkpoint) and then

re-running the component. Such techniques enable applications to recover from non-deterministic or transient errors, such as hardware failures, but cannot assist applications that fail due to deterministic errors. In contrast, Bolt's escape approach is designed to enable applications to recover from deterministic infinite loops.

The STEM [36] and ASSURE [35] systems are designed to enable applications to recover from deterministic failures. Each system instruments an application to take periodic snapshots of its state. On detection of a fault, the systems 1) record the previous snapshot of the program along with the current execution context at the site of the error and then 2) terminate the program. Offline, the systems then, via testing, identify a recovery strategy that returns an error code for a function on the stack at the time of the failure. The systems then insert this strategy back into the program to enable future executions to recover from the specific failure.

Bolt's unwind escape strategy is inspired by the techniques of these two systems. However, Bolt differs from these systems primarily in that it is designed to allow an application to recover the first time it experiences a failure. And, more generally, these systems are designed to deal with a different class of errors (i.e, memory safety).

The Rx system [32] takes periodic checkpoints of the program state. When a failure occurs, it 1) reverts the program to the checkpoint, 2) makes semantically safe changes to the application's execution environment (e.g., increasing the size of allocated buffers), and 3) restarts the execution. Rx has been shown to enable programs to recover from both non-deterministic and deterministic failures that are correlated with the execution environment. Like Rx, Bolt combines checkpoints with recovery actions. A difference is that escaping infinite loops may take the program outside its anticipated execution envelope (unlike the safe Rx recovery actions). We believe that such an approach is inherently required to eliminate infinite loop errors. Bolt implements a different set of recovery actions and additional binary analysis functionality (such as determining the loop structure) that is required to implement these recovery actions.

Failure-Oblivious Computing is an approach that compiles the source code of a program to add additional functionality that enables the application to dynamically

identify and recover from out-of-bounds memory reads and writes [34]. If the application detects that it is about to read or write an out-bounds-memory location, then it will synthesize a value for the read or expand the bounds of the allocated array to encompass the write. Unlike Bolt, Failure-Oblivious Computing, as proposed, requires the source code of the application and imposes a potentially high overhead (up to 8x [34]) on the normal execution of the program. However, FOC's approach is complementary in that it presents another alternative strategy for steering the program out of an infinite loop.

Researchers have also proposed a number of techniques that automatically repair programs by statically manipulating an application's code (or scheduler in the case of deadlocks) [25, 38, 31, 39, 24]. These approaches differ from that of Bolt in that they do not allow an application to recover on demand. Instead, these programs generate repairs after observing the first failure and, therefore, cannot be used to recover an output from a failed application.

6.3 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 provides a Force Quit Applications user interface and 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 [27]. However, these facilities usually offer only termination of a long-running task, while Bolt allows for the potential continued execution after the long-running loop subcomputation. In fact, the Bolt user interface is closely modelled after these task management interfaces. Extending these systems to work with Bolt would provide the user with the additional option of detecting and escaping infinite loops in unresponsive or long-running applications.

Chapter 7

Conclusion

Infinite loops can prevent programs from producing useful output, or cause users to lose important data. In more severe cases, an infinite loop which can be triggered remotely presents a security vulerability, enabling malicious users to launch denial of service attacks, and prevent others from accessing services. In most cases, users attempting to repair a broken program while still executing will not have access to the program source, and do not want to lose more work as the result of attempting to repair their program. Bolt addresses both of these issues. Bolt includes a mechanism for the detection of infinite loops in binaries, and a separate mechanism for escaping these loops. In addition, the Bolt interface combines each of these mechanisms into one tool, and adds higher level functionality, including an ability to search amongst possible escape strategies for a successful one.

Through the detailed case studies and evaluation, we have shown that infinite loops can be identified in practice in binaries, and that it is possible to find paths of continued execution that provide useful output, even without having access to the program source code. In twelve of the thirteen infinite loops analyzed, the Bolt Detector was able to identify an infinite loop. In all cases, the Bolt Escape tool was at least as good as terminating the program, and in all cases, one or more escape strategies gave a result at least partially equivalent to one later implemented by developers. The use of application checkpointing enables Bolt to provide several possible continued executions to the user, and select the one that delivers the most

desirable result.

The framework of the Bolt tool easily provides areas for continued work. The Bolt eetector tool currently detects only infinite loops that enter repeating states. With more complex analysis, other types of execution errors could be detected, for example, long running loops (a for loop with an unreasonable number of iterations), infinite or exponentially long recursion, deadlocks, etc. The Bolt escape tool, as currently implemented, would easily extend to additional strategies. The checkpointing functionality could be replaced with an arbitrary, perhaps more powerful alternative library.

Ultimately, Bolt attempts to be part of every user's toolbox, available to dispatch when faced with programs that are unresponsive. The existence of infinite loops and other bugs in modern software is inevitable. While the long term solution is generally a developer fix of the application, this offers little help to a users that need results immediately. The Bolt tool is built to provide this immediate help, and in some cases, produce the exact result that a fixed application would provide.

Appendix A

Dynamic Call Stack Analysis

A.1 Motivation

When first attaching to a running application, the Bolt Detector records the address of the current instruction. However, the next time the program returns to this instruction, it may be in a different stack frame. The algorithm described enables the detection tool to identify when one loop iteration has actually occurred by identifying if the instruction has been reached again in the same stack frame. This is accomplished using a relatively light weight instrumentation of function calls and returns from functions.

Consider the code shown in Figure A-1. Assume that the detector attaches to the application while the instruction pointer is at line 3. The state of the call stack is unkown to the detector at this point. However, we will assume in this case the call stack contains return addresses, and in this case the return addresses 16 :: 9. As execution continues the program will return from function `a()`, then function `b()`, then call function `a()` again at line 16. When reaching line 3, the call stack will contain the return address 17. If the detection tool takes no measures to keep track of the call stack state, at this point it will be unable to determine if a complete iteration of the loop has completed.

```
1.  void a()
2.  {
3.    return;
4.  }
5.
6.  void b()
7.  {
8.    a();
9.  }
10.
11. int main()
12. {
13.   while(1)
14.   {
15.     b();
16.     a();
17.   }
18.   return 0;
19. }
```

Figure A-1: Simple Demonstration Program

A.2 Explanation of Algorithm

In this section we reiterate the algorithm described in Chapter 3, and show its use on the demonstration program. Once the detector attaches to target program, calls to and returns from functions are instrumented. While monitoring these instructions, two partial call stacks are reconstruced dynamically. The first of these, S_c, represents the current state of the call stack. The second, S_i, represents the state of the call stack at the initial position. Both call stacks are initially empty. Each is modified according to the following rules, and contains function return addresses.

1. If the previous instruction was a return from a function, check if S_c is empty. If so, insert the current instruction address at the bottom of stack S_i, as if it was a queue.
2. If the previous instruction was a function return, and S_c is not empty, pop the top element off the S_c stack.
3. If the current instruction has the same address as the initial instruction, compare S_c and S_i. If they are the same, one loop iteration has completed.

4. If the current instruction is a function call, push the return address of this function onto S_c.

Assume the detector again attaches to the demonstration program at line 3 with the same state of the call stack. At this point, both S_c and S_i are empty. Table A.1 shows how these stacks change during the program execution.

Line	Actual	S_c	S_i
3	16::9	–	–
9	16	–	9
16	–	–	16::9
3	17	17	16::9
17	–	–	16::9
15	–	–	16::9
8	16	16	16::9
3	16::9	16::9	16::9

Table A.1: Call Stacks During Execution

Several points are noticeable by observing how each call stack changes. First, S_c is altered by the same rules as the actual call stack, except that elements are not popped off if it is already empty. Thus it is always equal to either the entire or part of the end of the actual call stack. Secondly, as referenced in Rule 1, when S_c is empty after a function returns, execution reaches a stack frame the detector has never seen. In these cases, it is certain the original instruction was called from this stack frame, and this address is added to S_i. Lastly, note the first time execution returns to line 3, $S_c \neq S_i$, and correctly, an iteration of the loop has not completed.

A.3 Proof of Correctness

First, we state precisely what the theorem we intend to prove is. While the end result of this analysis is to determine when returning to instructions if a loop iteration has completed, we restate this result in terms of the state of the call stacks at this instruction.

Theorem. *Consider a detector which attaches to an application at address A. At*

this point, the call stack has state S, the actual call stack. If the detector returns to an instruction at address A again, with some new call stack S', it will be possible to determine if $S = S'$.

To prove this theorem we will make use of two lemmas, each addressing one of S_c and S_i. These represent reconstructed partial call stacks. The actual current state of the entire current call stack during execution will be represented by C, and the actual state of the entire initial call stack when the detector attached will be represented by I.

Lemma 1. *If S_c has size k, then the last k elements of C will always be equal to S_c, in order.*

Proof. Initially, S_c is empty, and the match is trivial.

Assume after some sequence of modifications to C and S_c the property still holds. If the next instruction is a call instruction, the same address will be pushed onto both stacks, and the property will still hold. If the next instruction is a function return, the property will still hold if S_c is non-empty, as the same address will be popped off of both stacks. If S_c was empty, it will remain empty after the function return, and the property still trivially holds.

These are the only modifications made to C or S_c, thus by induction, this property holds at all times. □

Lemma 2. *If S_i has size k, then the last k elements of I will always be equal to S_i, in order. Moreover, if we assume C and I are intially equal, and have length n, immediately after an element is inserted into S_i, C will have length $n - k$, with the first $n - k$ elements of C matching with I.*

Proof. Initially, the property holds trivially, as $k = 0$. Since the stack I does not change during execution, we need only prove the property holds immediately after an element is inserted into S_i.

Consider the first time an elements is inserted into S_i. Before this, the number of call instructions seen must always have been at least as many as the number of

return instructions, as otherwise, S_c would not have been empty (as required in Rule 1). The address inserted into S_i will be equal to the address that was popped from the top of C.

Before this point, no modifications were made to the first n elements of C. Thus, the element of C popped off and inserted into S_i will be equal to the last element of I.

Assume the property holds after k insertions are made into S_i. After this element has been inserted, another will not be inserted until we have again seen more function returns than calls for the first time, at which point S_c will be empty, and C will have length $n - (k + 1)$ after the last element is popped off. The property will continue to hold. □

Now, we can prove the theorem.

Proof. Consider when we return to the original instruction, at address A. We have two cases. If $S_c \neq S_i$, then by our lemmas, since S_c and S_i are equal to the last k elements of C and I respectively, we can trivially see that $C \neq I$.

Now consider if we find $S_c = S_i$. By our lemmas, we see that the last k elements of C and I must be equal. Furthermore, by our second lemma, after inserting the kth element into S_i, the first $n-k$ elements of C matched with those of I. If modifications to the first $n-k$ elements of C were made after this point, then S_c would have length greater than k. Thus, since the last k and first $n - k$ elements of C and I are equal, they are equal in their entirety. □

Appendix B

Bolt User Manual

B.1 Downloading and Installing

Bolt is distributed as a tarball. To begin using Bolt, extract the download into a directory. The Bolt distribution contains several directories:

- **src:** The source specific to the Bolt tools, including the detection tool, escape tool, user interface, and tests.
- **pin:** Binary instrumentation tool Pin (developed by Intel). The Bolt detection tool uses Pin.
- **libunwind:** Stack unwinding library for linux [28]. The escape tool requires the libunwind library for the unwind escape strategy.
- **scons:** Bolt uses the scons build system. This directory contains all necessary build dependencies.

B.1.1 Requirements

Ensure python is installed. Version 2.x should be compatible. Bolt is not compatible with python 3.x

Bolt includes a distribution of Intel Pin, version 2.8-37300. If you are installing on a 64 bit linux distribution, you will also need to install the ia32 development libraries for Pin to use. You can use `sudo apt-get install ia32-libs`.

Optionally, before building Bolt, you may install BLCR (Berkeley Lab Checkpoint Restart) [2]. This allows a user to checkpoint a process before beginning analysis and using Bolt to alter its execution. If a user is unsatisfied with the results of Bolt, the state of the application can then be restored. BLCR is available at: ftg.lbl.gov/projects/CheckpointRestart/. Currently, it is only compatible with linux kernels through 2.6.34 (use `uname -r` to check). Follow the directions provided by BLCR to install.

B.1.2 Building Bolt

To build the package, run `./scons.py` from the top level of the Bolt distribution. Bolt includes a distrubtion of libunwind, version 1.0.1. The build script will begin by compiling this library. It will not be installed on the system, but in a local Bolt directory. Optionally, you can install libunwind on the system, and Bolt should also compile.

If you are running on a 64 bit system, and would like to use Bolt on 32-bit applications, you'll need to build the 32-bit version of the tool. Before attempting this, make sure you have compiled and installed a 32-bit version of libunwind separately, or the build will fail. To build 32-bit Bolt binaries, use the build32 option: `./scons.py --build32`. This should build the 32-bit versions of the Bolt Pin tool, the escape tool, and test programs. There's no need to use this option if building Bolt on a 32-bit system, everything will be built in 32-bit mode by default.

B.1.3 Running Tests

Tests for Bolt can be run from the top level directory as `./bolt_tests`. This python script is located at `build-src/GUI/bolt_tests.py`. Each test application contains a different type of infinite loop. Successfully running these tests will ensure that Pin

functions correctly on the system, and that the Bolt tools have been built properly.

B.2 Running Bolt

To start Bolt, simply type `./bolt` from the top level directory. This is a link to a python script that contains the Bolt functionality. The GUI should show a list of processes, and allow you to select target processes and perform detection and escape. The buttons on the GUI provide all of the functionality for the Bolt tools.

- **Checkpoint:** Attempt to take a checkpoint of the selected process.
- **Kill:** Kill the selected process.
- **Inject Check:** If selected, when detection is run on a target process, Bolt will attempt to inject the checkpoint library, enabling checkpointing for this program.
- **Timeout:** How long to attempt to detect an infinite loop (in seconds), and how long to monitor escape.
- **Detect:** Run Bolt detector on selected process.
- **Escape:** Run Bolt escape tool on selected process, using escape method chosen from the drop down.
- **Search:** Execute each escape strategy in sequence, restoring from a checkpoint, and report all results to the user.
- **Restore:** Restore program from selected checkpoint in the log.

Logs are stored in the `logs/` directory, and named by process and tool type. If checkpointing is enabled, this directory will also include the checkpoint files.

B.2.1 Potential Problems

If you encounter a permission error when attempting to run the Bolt tests or trying to use Bolt on an arbitrary process, your kernel may have ptrace protection enabled. For example, Ubuntu 10.10 and later prohibit the use of ptrace on non-child processes. To re-enable this, allowing Pin and the escape tool to attach to arbitrary processes, enter the command `sudo bash -c 'echo 0 > /proc/sys/kernel/yama/ptrace_scope'`.

This will remain effective as long as the machine is running. Note, this feature has been turned off by default to protect against possible malware attacks. It is possible that enabling it will make your system more vulnerable to particular types of malware.

B.3 Developing and Extending Bolt

After being built, the distribution includes additional files and directories:

- **build-src:** Includes binaries of the Bolt detection and escape tools, and versions of the GUI.
- **build-libunwind:** Includes libraries used to build the escape tool.
- **logs:** All logs when using Bolt tools are stored here.
- **bolt:** This file links to an executable script that starts the Bolt UI.
- **bolt_tests:** This executable script tests functionality of the detection and escape tools.

In addition, all the source code in `src/` is clearly divided into each of the Bolt components.

B.3.1 Adding Tests

To add additional tests, create a new file in the test directory, `src/Tests/`. Be sure to add the test to the SConscript file. The test must also be added to the `test_list`

file. In bar separated format, each line must include:

1. Name of the test
2. If the Bolt Detector tool should detect an infinite loop (y/n).
3. Which escape strategy to use (break/unwind)
4. If using unwind strategy, what return value to use
5. The expected return value from the program after escaping

B.3.2 Python API

The Bolt Python API is used by the Bolt GUI to provide easier access to the detection and escape tools. The implementation of this API can be found in `src/GUI/bolt_common.py`. Here we give a summary of the arguments and return values for this API, which can be used by arbitrary python programs.

`detect(pid, timeout=10.0, docheck=False, verbose=False, root=False)`

- **pid:** The process ID to attempt infinite loop detection on.
- **timeout:** The time, in seconds, to attempt detection of an infinite loop. For large infinite loops, detection can take several seconds.
- **docheck:** A boolean which indicates if the detection tool should attempt to inject the checkpointing library before starting the infinite loop analysis, as described above.
- **verbose:** If True, prints additional debugging information to the console.
- **root:** If True, the detection tool executes with root privileges. Useful for detecting infinite loops in programs running as other users or as roo.

Returns: `(error, loop, threadwarning, checklibresult)`

- **error:** If an error occurs, contains a string describing the error. Otherwise, equal to None.

- **loop:** A boolean indicating if an infinite loop was detected.
- **threadwarning:** A boolean equal to True if the detector encountered I/O, synchronization, or other operations that might indicate a program will eventually change state and break from the loop.
- **checklibresult:** If the detection tool attempted to load the checkpointing library, this indicates if it was successful.

`escape(pid, timeout=10.0, strat=‘‘break", retval=None, verbose=False, root=False)`

- **pid:** Process ID to attempt escape on.
- **timeout:** How long to allow the program to continue running after executing the escape tool before killing the escape process.
- **strat:** Currently equal to either “break” or “unwind”, indicating the strategy for the escape tool to use.
- **retval:** When using the unwind strategy, used to specify a value to force as a function return value.
- **verbose:** If True, prints additional debugging information to the console.
- **root:** If True, the escape tool runs with root permissions.

Returns: `(error, retcode, signal)`

- **error:** If an error occurs, contains a string describing the error. Otherwise, equal to None. In the case of a timeout, this string is equal to “Timeout”.
- **retcode:** The return code or signal number that terminated the target process.
- **signal:** A boolean indicating if the return code represents a return value or signal number.

Bibliography

[1] Apache http server project. http://httpd.apache.org.

[2] Berkeley lab checkpoint/restart. https://ftg.lbl.gov/projects/CheckpointRestart/.

[3] Gnu awk. http://www.gnu.org/s/gawk/.

[4] Php. http://www.php.net/.

[5] Wireshark. http://www.wireshark.org/about.html.

[6] Infinite loop in sub/gsub. http://www.osdir.com/ml/gnu.utils.bugs/2002-10/msg00045.html/, 2002.

[7] apr_fnmatch infinite loop on pattern "/*/web-inf". http://issues.apache.org/bugzilla/show_bug.cgi?id=51219, 2011.

[8] Bug 53632 php hangs on numeric value 2.2250738585072011e-308. http://bugs.php.net/bug.php?id=53632, 2011.

[9] psutil: A cross-platform process and system utilities module for python. http://code.google.com/p/psutil/, 2011.

[10] Pygtk: Gtk+ for python. http://www.pygtk.org, 2011.

[11] Working draft, standard for programming language c++. www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, 2011.

[12] A.R. Bradley, Z. Manna, and H.B. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.

[13] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE'09*.

[14] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot : A technique for cheap recovery. In *OSDI*, 2004.

[15] George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 2006.

[16] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin Rinard. Detecting and escaping infinite loops with Jolt. In *ECOOP*, 2011.

[17] M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV*, 2002.

[18] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: beyond safety. In *CAV*, 2006.

[19] Kees Cook. 100% cpu utilization in pam_env parsing. `http://bugs.launchpad.net/ubuntu/+source/pam/+bug/874565`, 2011.

[20] Victor C. Zandy et. al. Process hijacking, 1999.

[21] Fred Fierling. Bug 5303 - infinite loop in zcl discover attributes dissection. `http://bugs.wireshark.org/bugzilla/show_bug.cgi?id=5303`.

[22] Jim Gray. Why do computers stop and what can be done about it?, 1985.

[23] A. Gupta, T.A. Henzinger, R. Majumdar, A. Rybalchenko, and R.G. Xu. Proving non-termination. In *POPL*, 2008.

[24] H. Jula, P. Tozun, and G. Candea. Communix: A framework for collaborative deadlock immunity. In *DSN*, 2011.

[25] Horatiu Jula, Daniel Tralamazza, Cristian Zamfir, and George Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *OSDI*, 2008.

[26] Chi-keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[27] C. Metz. Mozilla girds firefox with 'hang detector', June 2010. `http://www.theregister.co.uk/2010/06/10/firefox_hang_detector/` .

[28] David Mosberger. The libunwind project. `http://www.nongnu.org/libunwind`, 2011.

[29] H.H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *International Symposium on Memory management*, 2007.

[30] David Patterson and et al. Recovery oriented computing (roc): Motivation, definition, techniques. Technical report, 2002.

[31] J. H. Perkins and et al. Automatically patching errors in deployed software. SOSP, 2009.

[32] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *SOSP*, 2005.

[33] Rick Regan. Why "volatile" fixes the 2.2250738585072011e-308 bug. `http://www.exploringbinary.com/why-volatile-fixes-the-2-2250738585072011e-308-bug`, 2011.

[34] 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 *OSDI*, 2004.

[35] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. Keromytis. Assure: automatic software self-healing using rescue points. In *ASPLOS*, 2009.

[36] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *USENIX Technical*, 2005.

[37] 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.

[38] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. ICSE'09.

[39] Jingyue Wu, Heming Cui, and Junfeng Yang. Bypassing races in live applications with execution filters. In *OSDI*, 2010.