MIT Libraries | DSpace@MIT

MIT Open Access Articles

Bolt: on-demand infinite loop escape in unmodified binaries

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Kling, Michael, et al. "Bolt: on-demand infinite loop escape in unmodified binaries." ACM SIGPLAN Notes 47,10 (2012), 431–450. https://doi.org/10.1145/2398857.2384648

As Published: 10.1145/2398857.2384648

Publisher: Association for Computing Machinery (ACM)

Persistent URL: https://hdl.handle.net/1721.1/125785

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

Bolt: On-Demand Infinite Loop Escape in Unmodified Binaries

Michael Kling Sasa Misailovic Michael Carbin Martin Rinard
MIT CSAIL
{mkling,misailo,mcarbin,rinard}@csail.mit.edu

Abstract

We present Bolt, a novel system for escaping from infinite and long-running loops. Directed by a user, Bolt can attach to a running process and determine if the program is executing an infinite loop. If so, Bolt can deploy multiple strategies to escape the loop, restore the responsiveness of the program, and enable the program to deliver useful output.

Bolt operates on stripped x86 and x64 binaries, dynamically attaches and detaches to and from the program as needed, and dynamically detects loops and creates program state checkpoints to enable exploration of different escape strategies. Bolt can detect and escape from loops in off-the-shelf software, without available source code, and with no overhead in standard production use.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Error handling and recovery; D.2.7 [*Distribution, Maintenance, and Enhancement*]: Corrections

Keywords Bolt, Infinite Loop, Error Recovery, Unresponsive Program

1. Introduction

Infinite and long-running loops can cause applications to become unresponsive, thus preventing the application from producing useful output and causing users to lose data.

1.1 Bolt

We present Bolt, a new system for detecting and escaping from infinite and long-running loops. Bolt supports an *on-demand usage model*—a user can attach Bolt to a running application at any point (typically when the application becomes unresponsive). Bolt will then examine the execution of the application to determine if it is in an infinite loop.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10. . . $10.00

Once it detects a loop, Bolt (directed by the user) tries multiple loop exit strategies to find a strategy that enables the application to escape the loop and continue to execute successfully. If one strategy fails to deliver acceptable continued execution, Bolt uses checkpoints to restore the application state and try another loop exit strategy. After the application has successfully escaped from the loop, Bolt detaches from the application.

To support the on-demand usage model, Bolt operates on unmodified, stripped x86 or x64 binaries. This makes it possible for Bolt to detect and escape loops in off-the-shelf software without available source code. To use Bolt, the user does not the need to recompile the application to insert instrumentation and Bolt does not incur any overhead in standard production use. Bolt can be installed and deployed even after the application has entered a loop.

Escaping the loop enables the application to continue executing and produce a result. This result may contain a part of or even all of the result that a (hypothetical) correct application (without the infinite loop) would have produced. The user can evaluate the effects of escaping the loop and can also, at his or her discretion, try several alternative strategies to recover the application execution from the loop, at the end selecting the result that best corresponds to his or her goals.

1.2 Bolt Workflow

Bolt consists of three components: a detector module, an escape module, and a checkpoint module.

The Bolt detector module first runs a dynamic analysis to find 1) the sequence of instructions that comprise one iteration of a loop and 2) the stack frame of the function that contains the loop. The detector module then monitors the execution of the application to determine if the loop is infinite: each time the application starts a new iteration of the loop, the detector takes a snapshot of the current application state and compares that snapshot to the last N-1 snapshots it has taken. If one of the previous snapshots matches, Bolt has detected an infinite loop.

After the detector module finds a loop (infinite or otherwise) a user may decide to invoke the escape module to continue the execution outside of the loop. A user may escape

from any loop that he or she perceives to be unresponsive (regardless of whether or not the detector module recognized the loop as infinite).

The Bolt escape module obtains (from the detector module) the set of *top-level instructions*—the set of instructions that are both in the loop and also in the function that contains the loop. The escape module can then explore alternatives from the following two escape strategies:

- **Jump Beyond Loop (Break):** The escape module sets the program counter to point to the instruction immediately following the top-level instruction with the highest address. This instruction is outside the set of instructions that comprise the loop—conceptually, it is the next instruction following the loop.
- **Return From Enclosing Procedures (Unwind):** The escape module exits either the function that contains the loop or any ancestor of this function that is on the current call stack. Each function on the call stack gives Bolt a distinct escape alternative.

Each escape strategy must also choose an *escape instruction* — an instruction inside the loop from which to escape the loop. Because the state may be different at different escape instructions, different escape instructions may produce different results. Bolt is currently configured to escape from the jump instruction at the end of the loop — i.e., the jump instruction with the highest address among all of the jump instructions in the top-level instructions. The Bolt implementation also has the capability to escape from any other top-level instruction.

There is, of course, no guarantee that any specific escape strategy will enable the application to continue to execute successfully. Bolt therefore offers the user the capability to explore multiple different escape alternatives. To provide this exploration, Bolt offers the option to checkpoint the application's state and then automatically apply each escape strategy in turn—if the user does not like how the application responds to one strategy, Bolt can roll back to the checkpoint and try another strategy.

1.3 Technical Challenges

Our previous work on Jolt [24] was the first to demonstrate a tool capable of detecting and escaping infinite loops at runtime. Jolt's mixed static and dynamic analysis approach limited its applicability in that it forced developers to recompile their applications with Jolt to enable loop detection and escape. This had two negative consequences: 1) users could not apply the technique on-demand (Jolt was available only for binaries precompiled to use Jolt) and 2) Jolt's inserted instrumentation imposed a 2%-10% overhead on the application even when Jolt was not running. Jolt also worked only for single-threaded applications.

Bolt, in contrast, operates on unmodified binaries, incurs no overhead during normal program execution, and works on both single-threaded and multithreaded applications. To realize Bolt's fully dynamic approach to infinite loop detection and escape on stripped binaries, we had to deal with the following key technical challenges:

- **Dynamic Loop Structure Detection:** The first step in detecting an infinite loop in a running application is to determine if the application is executing within a loop. Bolt implements a novel dynamic analysis to automatically reconstruct the loop structure in unmodified binaries.

  When Bolt attaches to an application, it records the current instruction. Bolt designates this instruction as the *pivot instruction*. Bolt identifies the loop by monitoring all executed instructions and tracking the sequence of function calls (via corresponding stack frames) between two subsequent executions of the pivot instruction. Note that the instructions in the loop may belong to multiple functions or even dynamically loaded libraries. Bolt detects the loop when the execution hits the pivot instruction in the same stack frame as when it first encountered the pivot instruction. The instructions executed in between these two executions of the pivot instruction comprise one iteration of the loop.
- **Exploration of Multiple Escape Alternatives:** In general, different escape strategies may produce different results for different applications. Bolt therefore enables the user to explore multiple escape strategies which continue the execution at different locations in the application. Bolt uses a checkpoint to restore the application to the state before any attempted escape, after which the other escape strategies can be applied to fix the execution.
- **Multithreaded Applications:** Many interactive applications are multithreaded—for example, applications often run the user interface and the core computation in separate threads. Bolt detects infinite loops in multithreaded programs on a per-thread basis. Bolt tracks memory accesses for loop iterations in every thread separately, but to better understand the synchronization behavior of threads, Bolt also tracks other actions, such as writes to shared memory and atomic instructions.

1.4 Experimental Results

We applied Bolt to thirteen infinite loops and two long-running (but finite) loops in thirteen applications to evaluate how well Bolt's detector module detects infinite loops and how well Bolt's escape module enables an application to continue its execution and produce useful results.

Detection. Bolt correctly classifies eleven of the thirteen infinite loops as infinite.[1] Bolt also correctly detects that the two long-running loops change state at every iteration and therefore reports that the loops may not be infinite.

[1] We note that *Jolt* can detect seven of these eleven infinite loops. The remaining infinite loops are out of the scope of Jolt.

Escape. For fourteen of the fifteen total loops (both infinite and long-running), Bolt can identify an escape strategy that enables the application to terminate and produce an output. These results include the two infinite loops that the detector module could not identify as infinite; we treated these two loops as additional long-running loops for this experiment.

For seven out of these fourteen 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 seven loops, Bolt enables the application to provide more output than simply terminating the application would produce, but not all of the correct output (see Section 4 for details).

1.5 Infinite Loop Patterns

An examination of the infinite loops in our set of benchmark applications indicates that they fall into one of two patterns: *missing transition* infinite loops (the application enters a state from which it cannot consume and process the remaining input) and *missing exit condition* infinite loops (the application has processed its input but is missing the exit condition that enables it to terminate after processing that particular input).

These patterns help to explain why Bolt can enable applications to produce useful or even correct results. For missing transition loops, using Bolt to escape the infinite loop typically enables the application to produce a partial result (in some cases escaping the loop restores the ability of the application to successfully process subsequent inputs so that the application produces the correct result). For missing exit condition loops, using Bolt to escape the infinite loop typically enables the application to produce the correct result.

1.6 Contributions

This paper makes the following contributions:

- **Bolt:** We present Bolt, which detects and escapes infinite loops in (potentially stripped) single and multithreaded x86 and x64 binaries. Bolt uses application checkpointing to explore multiple escape strategies. The goal is to enable the program to continue its execution to provide useful results to its users even in the face of infinite loops.
- **Implementation:** We present the details of the Bolt implementation, including the mechanisms Bolt uses to 1) detect loops in executing binaries, 2) determine if the loop is actually an infinite loop, 3) escape the loop if directed to do so by the user, and 4) roll back to the checkpointed state to try different alternatives if the last alternative did not deliver a successful execution.
- **Evaluation:** We present an evaluation of how well Bolt works for thirteen infinite loops and two long-running loops in thirteen applications. Bolt's infinite loop detection is successful for eleven out of the thirteen infinite loops. Escaping the loops enables the applications to successfully continue their executions for fourteen out of the fifteen total loops. For seven of these loops Bolt produces the same correct result as subsequent versions of the applications with the infinite loops eliminated via a developer fixes. For the remaining seven loops, Bolt provides more useful output than simply terminating the applications.
- **Case Studies:** We present five case studies that illustrate, in detail, the effect of using Bolt to escape infinite loops.
- **Infinite Loop Patterns:** We discuss two infinite loop patterns found in our set of benchmark applications: *missing transition* infinite loops and *missing exit condition* infinite loops. These patterns help to explain Bolt's success in enabling applications to successfully execute once they escape infinite loops.

We also note that Bolt currently implements the core system building blocks required to address a much larger range of anomalies and errors. Specifically, Bolt's monitoring, checkpointing, and search capabilities can be extended to respond to a variety of detected errors, enabling a user to try different recovery strategies until one allows the application to continue successfully.

2. Using Bolt

The PHP script in Figure 1 performs a computation, adds a small constant numerical value to the computed result (`$result`), and then outputs the result. A user might naturally expect the script to print its results and terminate normally. However, with PHP version 5.3.4, the PHP interpreter enters an infinite loop while processing the next to last line (which adds a double-precision floating point literal to `$result`).[2] The script hangs and never prints the result.

Bolt. If the user observes that PHP has become unresponsive, he or she can install and apply Bolt on-demand. To apply Bolt, the user would open the Bolt User Interface (presented in Figure 2) and perform the following steps:

- **Detect:** First, the user selects the `php` process from the central box that lists running processes, sorted by CPU usage; in this case the `php` process consumes almost 100% of the CPU time. The user next presses the "Detect" button, after which Bolt runs its infinite loop detection analysis and then reports in the status box below the list of processes that it has detected an infinite loop; Figure 3 provides a more detailed presentation of this box. Note that a user does not need to anticipate the existence of the infinite loop in advance—it is possible to install and apply Bolt even after the PHP interpreter becomes unresponsive.

[2] PHP 5.3.4 has a known infinite loop bug when parsing source code containing certain small floating point numbers or when string literals that represent these numbers are passed as an argument to a string-to-float conversion function such as `floatval()`. We discuss the technical aspects of this bug in Section 5.1.

```
<?php
   $result = 0
  // ...

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

Figure 1: Example PHP Program

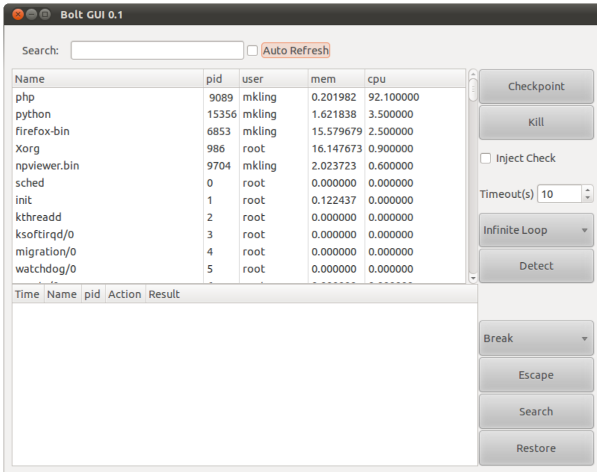

Figure 2: Bolt GUI Main Window

Time	Name	pid	Action	Result
22:29:17	php	5435	Escape	Strategy Break: Success
22:29:17	php	5435	Restore	Success
22:29:17	php	5435	Escape	Strategy Unwind: Error, Return Code 255
22:29:17	php	5435	Checkpoint	Success
22:29:17	php	5435	Detect	Detected: Yes. Checkpointing enabled.

Figure 3: Bolt Status Messages

- **Checkpoint:** After Bolt detects the infinite loop, the user has an option to immediately try one of the escape strategies or to take an application checkpoint by pressing the "Checkpoint" button. A checkpoint efficiently stores the entire state of the program and allows a user to try multiple escape strategies—if the first strategy does not give acceptable results, Bolt can use the checkpoint to restore the state and try some other escape strategy.
- **Escape:** The user chooses an escape strategy from the drop down menu above the "Escape" button and hits the "Escape" button. In our example, the user first selects the *Unwind* strategy (which exits from the function that contains the infinite loop). This strategy does not produce a desired result—the execution of PHP continues, but the program returns a non-zero status code and prints an error message reporting a parsing error on the line before the last (on which the small floating point value is added to the `$result` variable).
- **Restore:** If an escape strategy does not produce an acceptable result, then the user can press the "Restore" button to restore the application's original state from the checkpoint and execute a different strategy. In our example, the user restores the application and then attempts the *Break* strategy (which jumps to an instruction in the same function outside of the observed infinite loop). In this case the strategy is successful and the user can obtain the output of the script.

Automated Bolt. Instead of manually selecting an escape strategy, the user can press the "Search" button, which automates the process of executing escape strategies and restoring from checkpoints. After trying one escape strategy, Bolt checks if the application terminated gracefully (without error), and if so, presents the results to the user. If the program terminated unexpectedly, did not terminate until a timeout, or the user is not satisfied with the result, Bolt restores the program from a checkpoint and tries the next escape strategy. It repeats these steps until it finds a successful escape strategy or explores all of the different strategies.

When encountering a non-responsive program for the first time, we anticipate that users will typically elect to invoke an automated search for the best escape strategy. On the other hand, if a user encounters a loop in a program for which Bolt previously found a successful escape strategy, a user may select that strategy directly from the drop down menu.

3. Bolt Implementation

Bolt contains a detection module, an escape module, and a checkpoint module. These modules are tied together with the Bolt User Interface, but can also be used separately. Each of the Bolt modules is implemented using the following tools:

- **Bolt Detector:** The Detector attaches to the target application using the Pin dynamic instrumentation framework [34]. The Bolt Detector uses Pin to monitor memory accesses and function calls.
- **Bolt Escape Tool:** The Escape tool uses `ptrace` system calls to attach to the target application, set breakpoints, and execute escape strategies.
- **Checkpoint Module:** The checkpoint module is implemented using Berkeley Lab Checkpoint/Restart (BLCR), a Linux kernel module for application checkpointing [2].

In addition to providing these modules, Bolt is also a completely modular framework, enabling it to be extended with additional error detectors and escape strategies.

3.1 Bolt Detector

The Bolt Detector performs two primary tasks: *loop structure detection* and *infinite loop detection*.

3.1.1 Loop Structure Detection

When the Bolt Detector attaches to the running application using Pin, it must first detect if the application is inside a loop. A loop is defined as a sequence of executed instructions that starts from a given stack frame and instruction and

then returns to the same stack frame and instruction. This definition includes loops that may cross function boundaries through function calls, but excludes recursion.

When the Bolt Detector attaches to the running application, it saves the value of the instruction pointer at the moment it attaches; we call this instruction the *pivot instruction*. The Bolt Detector then tests if the application's execution is within a loop that contains the pivot instruction—specifically, if the application executes a contiguous sequence of instructions beginning from the current stack frame and the pivot instruction and returning to the same stack frame and the pivot instruction.

To determine if the stack frame of a subsequently executed instruction is the same as the pivot instruction, the Bolt Detector compares the *calling context* of the instruction to that of the pivot instruction.[3] A calling context is the sequence of executed call sites that create a given stack frame.

The Bolt Detector dynamically reconstructs the calling context of both the pivot instruction and the instructions in the loop by monitoring the execution of the application.

Calling Context Reconstruction. The Bolt Detector's calling context reconstruction algorithm exploits the observation that it is possible to construct the calling context of the pivot instruction by observing the subsequent execution of the application.

To make this more precise, we first abstract the execution trace of the application after Bolt attaches as a sequence of only function calls and returns. The function call instruction i : `call f` is an instruction at address i that pushes the address $i + 1$ onto the application's stack and then transfers control to the address of the function `f`. The function return instruction `ret` is an instruction that pops the return destination address off the top of the application's stack and then transfers control to the return destination.

We classify a trace of the execution of the application as *matched* if the trace is an element of the language of balanced parentheses with grammar M:

$$M := \epsilon \mid (\ M\ )\ M$$

In this grammar, left parentheses "(" denote `call` instructions and right parentheses ")" denote `ret` instructions. This grammar therefore represents executions where each function call instruction is appropriately matched by a function return instruction.

The calling context of a pivot instruction in an execution trace is the given by the sequence of *unmatched* return instructions in the suffix of the trace beginning at the pivot instruction. The suffix of a matched trace can be unmatched in that it may contain return instructions ")" with no corresponding call instruction "(". If a return instruction is unmatched in this suffix, then the corresponding call instruction must have occurred before the pivot instruction and is therefore part of the calling context.

[3] The option to compile an application without a stack frame pointer is available in many widely used compilers. Therefore, the stack frame pointer cannot be reliably used to identify stack frames.

Algorithm 1 presents the pseudocode for the calling context reconstruction algorithm. The algorithm updates the calling context information for the pivot instruction after each subsequent instruction step as follows. It takes as input the current instruction address i_c, the current stack pointer sp, the current call stack S_c, and the portion of the pivot instruction's calling context that has been constructed thus far C_p. In this presentation, S_c is a standard stack data structure whereas C_p is a standard list.

Algorithm 1 Calling Context Reconstruction

```
1: function UPDATE_CONTEXT (i_c, sp, S_c, C_p)
2:     if is_call ( i_c ) then
3:         push ( S_c, i_c + 1)
4:     else if is_ret ( i_c ) then
5:         if empty ( S_c ) then
6:             insert ( C_p, *sp )
7:         else
8:             pop ( S_c )
9: end function
```

The algorithm need only consider three cases:

- **Call Instruction:** If the current instruction is a call instruction, then the algorithm pushes the instruction address plus one onto the current call stack of the application S_c (Line 3).
- **Unmatched Return Instruction:** If the current instruction is a return instruction, and the current call stack is empty, then the return is unmatched and therefore part of the calling context. The algorithm adds the return destination address to the calling context (which resides at the top of the stack and is pointed to by sp) by inserting it into the calling context C_p (Line 6).
- **Matched Return Instruction:** If the current instruction is a return instruction and the current call stack is non-empty, then the return is matched and not part of the calling context. The algorithm therefore pops the top element off of the current call stack (Line 8).

Each time the Bolt Detector reaches the pivot instruction it compares C_p with S_c. If these two structures contain the same sequence of addresses, then both the calling context of the pivot instruction and the current call stack are the same and the Bolt Detector has identified a loop in the execution of the program.

Applicability. The loop structure detection algorithm identifies loops implemented using standard control flow constructs (both structured and unstructured) such as for and while loops, conditionals, goto statements, and function call/return. It is not designed to recognize loops generated using a mix of `setjmp`/`longjmp` and/or recursive function calls. Note that the application must re-execute the pivot instruction for the algorithm to detect the loop.

3.1.2 Infinite Loop Detection

At the end of each iteration of the loop, the Bolt Detector takes a snapshot of the application's register and memory state. These snapshots 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 (as recorded in the snapshots), then the loop is infinite. The Bolt Detector records any memory locations modified or read in any previous loop iteration. When comparing snapshots, the Bolt Detector first compares the set of memory locations and registers contained in the snapshots. If these sets are equal, it then compares the contents of these memory locations and registers.

The Bolt Detector also has the capability to detect certain *semantic* infinite loops in cases where changes to portions of application's state are independent of the termination of the loop. For example, an infinite loop may produce outputs to a console or file during each iteration through calls to the application's runtime library (e.g., `fputs` or `fprintf` in the C standard runtime library). On each of these calls, the standard library keeps track of the number of bytes written or the current cursor position in the file in its internal state, which is stored within the application's address space. While this state will change on each iteration of the loop, if this state is never used within an iteration of the loop, then it can be ignored (its value is independent of the termination of the loop). To implement this functionality, the Bolt Detector uses Jolt's library routine abstraction technique [24].

In some programs that contain infinite loops, it may take many iterations before the program returns to an earlier state. The Bolt Detector provides a user-settable timeout, or a minimum number of instructions to execute, after which it will detach if it has not yet detected an infinite loop.

Multithreaded Loop Detection. When the Bolt Detector 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 to determine if there may be an infinite loop in that thread. In addition to memory snapshots, the Bolt Detector tracks other actions taken by each thread before reporting that there is an infinite loop (it is possible for another thread to eventually take some action to influence the execution of a thread suspected to be in an infinite loop). Specifically, the Bolt Detector monitors each thread for the following types of operations, which indicate that threads may be communicating with each other:

- **Synchronization/Atomic Operations:** The Bolt Detector inspects the set of instructions executed by each thread to identify synchronization and atomic operations.
- **Volatile/Shared Memory Writes:** The Bolt Detector compares the read and write sets of each thread in the application to find intersections between the read set of a potentially looping thread and the write set of another thread.

If the Bolt Detector observes any potential inter-thread communication, then it reports to the user that the thread may not be in an infinite loop (however, it still provides the user with the capability to escape the loop).[4]

3.1.3 Analysis Result

The Bolt Detector's produces 1) the set of *top-level instructions* (the set of instructions that are both in the loop and also in the function that contains the loop) along with the stack pointer observed at each instruction, 2) a determination of whether the loop is potentially infinite, and 3) a report of any patterns that may indicate inter-thread communication. Before exiting, the Bolt Detector writes these results to a file and then detaches all dynamic instrumentation.

For loops that do not read inputs from the outside world and are within an application without asynchronous control flow (such as signal handlers) or multithreading, the Bolt Detector's infinite loop detection strategy is sound—if it reports that there is an infinite loop then the application is in an infinite loop. If any of these qualifications are violated, then the detection strategy takes a best-effort approach by looking at additional information (such as the application's synchronization behavior). In these cases, the Bolt Detector reports that the application may not be in an infinite loop, but still offers the user the option to escape the loop.

3.2 Bolt Escape Tool

The Bolt Escape tool implements two infinite loop escape strategies: *Break* (which forces the execution to continue at an instruction that is outside the set of instructions executed by the loop) and *Unwind* (which forces the execution to return from the function containing the loop).

The Escape tool takes as input the results of the Bolt Detector's infinite loop detection analysis and the escape configuration parameters provided by the user via the Bolt GUI. If the user has elected to use the Unwind strategy, then the Escape tool also takes as input the value to return from the target function when unwinding the stack.

Given the analysis results and GUI parameters, the tool proceeds as follows:

1. The escape tool first attaches to the application and places a breakpoint at the *escape instruction*—the instruction from which escape will be performed. The Escape tool currently selects the top-level instruction with the maximum address as the escape instruction.
2. When the application hits the breakpoint, the escape tool compares the current stack pointer of the application with that provided as input with the top-level instructions. This ensures that the breakpoint has been hit in the top-level stack frame. If not, execution continues until the application hits the breakpoint in the correct stack frame.

[4] Bolt's analysis is inspired by the C++0x standard's definition for "empty" loops that can be removed by an optimizing compiler [18].

3. When the application reaches the breakpoint in the correct stack frame, the escape tool applies one of the two escape strategies and then remains attached to the application for a specified timeout. If the application terminates within this timeout, then the Escape tool reports the termination to the user. Otherwise, the Escape tool detaches from the application once the timeout expires, allowing the application to continue its execution with no instrumentation and no Bolt overhead.

The search function in the Bolt GUI provides an automated method to explore multiple escape strategies on a target program. To use the search function, a user must attach Bolt to the program and take a checkpoint. The search function will then try each escape strategy in turn. For each strategy, if the target program does not terminate after a timeout, the user can attempt to determine if the escape strategy was successful by observing the continued execution. If the strategy did not succeed, Bolt will terminate the program (if necessary), then restore execution from the checkpoint.

Implementation. The Escape tool attaches to the application with `ptrace`, which is a lightweight alternative to attaching to the application with Pin in cases in which one does not need to perform heavyweight dynamic instrumentation.

The Escape tool implements the Break escape strategy by using `ptrace` to modify the instruction pointer register of the application to point to an *escape destination instruction*. This action forces the application to immediately jump to the escape destination instruction. The Escape tool selects the address of the escape instruction plus one as the escape destination instruction. Because we set the escape instruction to be the top-level instruction with the maximum address, the escape destination is therefore not in the set of top-level instructions and must be on a different control flow path. For example, if the escape instruction is a conditional jump, then the escape destination is the fall-through of the jump. Bolt also supports setting the escape instruction to be any top-level jump instruction.

The Escape tool uses the libunwind library [6] to implement the Unwind escape strategy, which unwinds one or more frames of the stack, effectively forcing the program to return from the current function and continue execution after the function call-site. The Bolt GUI also enables a user to explore different return values for the function call when executing the Unwind strategy. The Escape tool implements this by modifying the return value registers immediately before unwinding the application from the function.

The libunwind library uses a combination of exception handling information and stack walking to perform call stack reconstruction. In particular, libunwind looks at certain sections in ELF binaries, including the `.eh_frame`, and `.debug_frame` sections. Depending on how the application was compiled, these sections may not contain enough information to reconstruct a call stack. In these cases, libunwind uses a simple stack walking algorithm that looks at the base pointer. In general, libunwind's analyses are incomplete. Therefore, in cases where libunwind is unable to identify an unwind destination, Bolt reports to the user that is is unable to apply the Unwind strategy.

3.3 Checkpoint Module

Bolt uses the BLCR (Berkeley Lab Checkpoint/Restart system [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.

One limitation of BLCR is that it requires an application to have loaded the BLCR runtime library into its address space before creating a checkpoint. To enable Bolt's on-demand usage scenario where the user downloads Bolt on the fly, we therefore automatically force-load BLCR's runtime library into the application's address space when the user elects to take a checkpoint. Specifically, we use Pin to inject calls to `dlopen()` into the application's instruction stream. This approach is similar to what others have done in previous work on Process Hijacking [52].

3.4 Platform Compatibility and Extensions

Bolt's current implementation best supports detecting and escaping loops in x86 and x64 binaries on Linux. In principle, it would be possible to extend Bolt for use on other platforms in the following ways:

- **Bolt Detector:** The Bolt Detector uses Pin for its dynamic analysis. Pin can analyze binaries built for the x86, x64, and IA-64 architectures on Linux, Windows, and Mac OS. To support an alternative architecture or operating system, one would need a different dynamic instrumentation framework (e.g., DynInst [29] or DynamoRIO [21]) that supported the desired platform.
- **Escape Module:** The escape module is x86/x64 Linux specific. To extend the escape module to an alternative architecture, one would need to provide Bolt with access to the application's stack pointer register. To extend the escape module to an alternative operating system, the operating system would need to provide functionality similar to that of `ptrace`. So, for example, while Mac OS X provides an implementation of `ptrace`, Windows does not. Therefore, an implementation of the escape module for Windows would need to obtain `ptrace`'s capabilities on that platform.
- **Checkpoint Module:** The checkpoint module is a Linux kernel module and is compatible with all versions of Linux up to and including version 2.6.38. The module fully supports x86 and x64 architectures, and has experimental support for ARM and PPC.

  To provide checkpointing on other architectures in Linux, one would need to re-implement the small architecture specific components of BLCR [3]. To provide check-

pointing on other operating systems, one would need to use an alternative application checkpointing framework that supported the operating system of interest.

4. Empirical Evaluation

We next present and discuss the benchmarks we use to evaluate Bolt, our experimental methodology, and the results of our experiments.

4.1 Benchmark Summary

To evaluate Bolt, we collected a number of applications with known infinite loops. We obtained eight infinite loops in ctags, grep, ping, look, and indent that we used in our previous evaluation of Jolt [24]. In addition to these infinite loops, we searched open-source project repositories and the Common Vulnerabilities and Exposures database [4] for additional bug reports that involve non-terminating program behavior.

We used several general guidelines while searching for additional bug reports to investigate. First, we only investigated bug reports that specified the steps required to reproduce the infinite loop and provided an input that elicited the infinite loop. Second, we attempted to collect infinite loops from different application domains. Finally, we considered the perceived importance and popularity of applications when determining which bug reports to investigate.

We investigated nine additional bug reports: five of these bugs are due to infinite loops, two are due to long-running loops, and one is due to an infinite recursion. We were unable to reproduce the remaining loop.

Benchmarks. Table 1 presents the benchmarks on which we evaluated Bolt. Column 1 (Benchmark) presents the name of the benchmark. Column 2 (Version With Bug) presents the version number of the benchmark that contained the infinite or long-running loop. Column 3 (Version With Fix) presents the version number of a later version with the bug eliminated via a developer fix. Column 3 (Loop Type) indicates if the benchmark loop is an infinite loop or a long-running loop.

We evaluate Bolt on fifteen benchmarks. Thirteen benchmarks contain infinite loops and two benchmarks contain long-running (but finite) loops. The applications range from common utilities to large, multithreaded GUI applications:

- **php:** PHP is a general purpose scripting language, commonly used for web development and in server side applications [8]. The infinite loop occurs when the PHP interpreter parses certain floating point values 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 these floating point value as input or parse a script that contains these values [17].
- **wireshark:** Wireshark is a network protocol analyzer which allows a user to capture and analyze traffic on a computer network [10]. The infinite loop occurs when parsing certain network packets that contain malformed attributes. The infinite loop can be triggered by opening a previously recorded packet log file or by a remote user sending corrupt network packets [14].
- **gawk:** Gawk is the GNU implementation of the AWK pattern scanning and text processing language [5]. The long-running loop occurs when attempting to parse an input containing nested single and double quotes. This loop occurs in the implementation of the regular expression engine in gawk [11].
- **apache:** Apache is an extensible web server [1]. The infinite loop occurs in the Apache Runtime Library in a function used for matching URL regular expressions [15].
- **pam:** Pluggable Authentication Modules (PAM) are used to provide authentication mechanisms for Linux programs [7]. The infinite loop is within the PAM module that is responsible for parsing files that contain environment variables. It can be triggered when parsing a file that has values that exceed the maximum internal buffer size [16].
- **poppler:** poppler is a rendering engine for the PDF document format [9]. It contains a long-running loop that occurs when rendering PDF documents that contain JPEG images with corrupt dimensions [13].
- **ctags:** We investigated two infinite loops in ctags:
  - **ctags-fortran:** The fortran module in version 5.5 has an infinite loop when parsing certain declarations separated by semicolons.

Benchmark	Version With Bug	Version With Fix	Loop Type
php	5.3.4	5.3.5	Infinite
wireshark	1.4.0	1.4.1	Infinite
gawk	3.1.1	3.1.2	Long
apache	2.2.18	2.2.19	Infinite
pam	1.1.2	1.1.3	Infinite
poppler	0.11.3	0.12.0	Long
ctags-fortran	5.5	5.5.1	Infinite
ctags-python	5.7b (646)	5.7b (668)	Infinite
grep-color	2.5	2.5.3	Infinite
grep-color-case	2.5	2.5.3	Infinite
grep-match	2.5	2.5.3	Infinite
ping	20100214	20101006	Infinite
look	1.1 (svr 4)	-	Infinite
indent	1.9.1	2.2.10	Infinite
java-vm	6.0	-	Infinite

Table 1: Benchmark Loops

- **ctags-python:** The python module in version 5.7 has 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 to print only the matching part of each line.
- **ping:** ping client is a network utility that checks for the reachability of a remote host on a network. An infinite loop occurs when parsing certain reply message fields.
- **look:** look is a dictionary lookup program. An infinite loop occurs as a result of a dictionary entry not terminated by a newline character.
- **indent:** indent is a source code pretty-printer. An infinite loop occurs parsing a final line containing a comment and no newline character.
- **java-vm:** Java virtual machine has an infinite loop that occurs when converting certain strings to floating point numbers [12].

Additional Infinite Loop Reports. In addition to the benchmark infinite loops discussed above, we also investigated the following infinite loop bug reports:

- **findutils:** The bug reported in GNU findutils (GNU bug tracker 13381) is an infinite recursion, not an infinite loop. It is therefore outside the scope of Bolt (although one can envision extending Bolt to handle infinite recursions as well as infinite loops). The infinite recursion is caused by symbolic references in the file system that form cyclic paths in scanned subdirectories.
- **gstreamer:** The bug reported in the gstreamer audio library (Gnome bug tracker 120292) is an infinite loop in the vorbis plugin. We were unable to compile the version of the plugin package that contains the infinite loop.

4.2 Methodology

For each benchmark, we performed the following steps:

- **Detection:** We ran the application and allowed it to enter the infinite or long-running loop. We then started Bolt and evaluated if Bolt was able to identify the infinite loop.
- **Checkpointing:** We evaluated whether Bolt (using the BLCR library) was able to take a checkpoint of the application and restore from a previous checkpoint. We evaluated the efficiency of checkpointing by measuring the time to take the checkpoint and size of the checkpoint.
- **Escape:** We used Bolt to force the application to escape from the loop. We then observed whether the application became responsive as a consequence of the escape. We determined whether the resulting output was well formed and used manual inspection or Valgrind to determine if the continued execution incurred any invalid memory accesses or leaks.
- **Comparison with Termination:** We compared the output obtained from terminating the application with the output obtained by applying each of the Bolt escape strategies.
- **Comparison with Manual Fix:** We compared the output produced by applying different Bolt escape strategies with the output of a later version of the application with the infinite loop manually corrected via a developer fix.
- **Comparison between Strategies:** We considered two possible loop escape strategies: Unwind and Break. We compared the effectiveness of each of these strategies by comparing the output produced after using each of these strategies.
- **Reasons for Infinite Loops:** We manually analyzed the source code of the application to discover the underlying reasons for the infinite loop and classify these reasons into common patterns.

We performed our experiments on a 3.2 GHz Quad-Core Intel Xeon with 6 GB of RAM running Ubuntu Linux. We also used the Linux `strip` command to remove all symbols from executables and relevant libraries before running Bolt.

We set two timeouts for how long to run the Bolt Detector: 10 seconds of execution time or 10^5 analyzed instructions; detection stops when one of the conditions is hit. We did not constrain the Bolt Detector to maintain a fixed number of snapshots—therefore, the number of snapshots that we compare varies from application to application. The Break escape strategy jumps out of the loop from the jump instruction at the maximum address inside the loop. For the Unwind escape strategy we used four possible return values: -1, 0, 1, and the value previously stored in the CPU's return register (e.g., the eax register on x86). We applied both detection and escape for up to three consecutive times for each unresponsive loop to account for possible infinite loops in the continued execution.

4.3 Detection Results

Bolt detected 11 out of 13 infinite loops. Bolt was not able to detect the infinite loops in indent and java-vm. For these two benchmarks the state of the application changes after every iteration of the infinite loop that Bolt tracked.

For 9 out of these 11 loops Bolt was able to identify a repeating state after a single iteration of the loop. Two loops, php and pam, change the state of the application after a single iteration, but the executions eventually return to one of the previously observed states after multiple iterations.

For the two long-running (finite) loops in gawk and poppler Bolt identifies that the state after every iteration changes and thus does not report them as infinite loops.

In comparison to Bolt's results, Jolt can successfully detect seven of these infinite loops (ctags-fortran, ctags-python, grep-match, grep-color, grep-color-case, ping, and look). The remaining benchmarks are out of the scope of Jolt, either because the benchmark is multithreaded (wireshark), because the loops execute several iterations before repeating the same state (php and pam) or because the loops occur in library code (apache, pam, and poppler).

4.4 Numerical Results

Timing Results. Table 2 presents the time statistics for infinite loop detection and escape. Column 2 presents the amount of time that Bolt takes to detect an infinite loop, starting from when the Bolt Detector has attached to the application. Column 3 presents the amount of time required to escape from the infinite loop and continue execution, starting from when the Bolt escape module has attached to the application. These times represent the median of five repeated executions in which we used Bolt to escape each infinite loop. The escape time for each benchmark is equal to the maximum of the times for the two escape strategies.

For all benchmarks, median detection time was below 6 seconds and the median escape time was below 0.03 seconds. Entries marked as "–" indicate that the Bolt Detector did not detect an infinite loop until the timeout.

Loop Size Results. Table 3 presents the length of the infinite loop and the size of the snapshot of the program state after each iteration for each benchmark for which Bolt detected the infinite loop. Column 2 presents the size of a register file in bytes (note that because php is a 32-bit application its register snapshot size is smaller than the remaining 64-bit applications). Column 3 presents the size in bytes of all memory locations modified during the execution of the loop. Column 4 presents the length of each detected infinite loop measured in the number of instructions that the loop executes. It is possible for different iterations of infinite loops in our benchmark applications to have different sized snapshots or lengths, depending on the location in the loop to which Bolt attaches. In such cases we reported the maximum sizes and lengths over all iterations of the loop.

We identified a positive correlation between the lengths of each loop and the time to detect the loop (R-squared coefficient 0.78). This positive correlation suggests that loops with more instructions will typically take longer to detect. The short detection time for the smallest loop in Wireshark, which is one instruction long (see Section 5.2), indicates that the overhead of the initial setup of Bolt Detector is minimal.

Checkpointing Results Table 4 summarizes the checkpointing results. Column 2 presents whether the checkpointing and subsequent restoring of the state was successful. Column 3 presents the median time to create a checkpoint.

Benchmark	**Detection Time** (s)	**Escape Time** (s)
php	1.346	0.008
wireshark	0.0003	0.006
gawk	–	0.004
apache	0.168	0.005
pam	5.856	0.011
poppler	–	0.002
ctags-fortran	0.122	0.004
ctags-python	0.141	0.002
grep-match	2.458	0.013
grep-color	2.145	0.011
grep-color-case	1.348	0.006
ping	0.026	0.003
look	0.164	0.003
indent	–	0.024
java-vm	–	0.030

Table 2: Loop Time Statistics

Benchmark	**Reg. Size** (b)	**Mem. Size** (b)	**Length**
php	128	688	7784
wireshark	192	0	1
gawk	–	–	–
apache	192	8	106
pam	192	0	74606
poppler	–	–	–
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
indent	–	–	–
java-vm	–	–	–

Table 3: Infinite Loop Memory and Length Statistics

Benchmark	**Success**	**Time** (s)	**Size** (kb)
php	Yes	0.221	2192
wireshark	No	–	–
gawk	Yes	0.139	484
apache	No	–	–
pam	Yes	0.269	943
poppler	Yes	1.663	94548
ctags-fortran	Yes	0.166	360
ctags-python	Yes	0.136	448
grep	Yes	0.147	408
ping	Yes	0.177	360
look	Yes	0.181	212
indent	Yes	0.736	228
java-vm	Yes	0.258	7116

Table 4: Results of BLCR Checkpointing

Benchmark	Errors After:		vs. Termination		vs. Manual Fix		Best
	Unwind	Break	Unwind	Break	Unwind	Break	Strategy
php	Parse†	None	Same	Better	None	Same	Break
wireshark	None	Parse†	Better	Better	Same	Same*	Same
gawk	None	Memory†	Better	Same	Partial	None	Unwind
apache	None	None	Better	Worse	Same	Worse	Unwind
pam	None	Unresponsive	Better	N/A	Same	N/A	Unwind
poppler	None	Unresponsive	Better	N/A	Partial	N/A	Unwind
ctags-fortran	None	Memory	Better	Same	Partial	None	Unwind
ctags-python	None	None	Better	Better	Partial	Partial	Same
grep-match	None	Unresponsive	Better	N/A	Partial*	N/A	Unwind
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
java-vm	Unresponsive	Unresponsive	N/A	N/A	N/A	N/A	N/A

Table 5: Summary of Escape Results

Column 4 presents the median size of the checkpoint. The results contain only a single entry for grep as all three infinite loops occur in the same version of the program.

For 10 out of the 13 applications, restoring from the checkpoint was successful. Restoring the state for Wireshark and Apache was not successful because the BLCR checkpointing system was unable to checkpoint open sockets and certain open system files (such as `/dev/urandom`).

The time to take the checkpoint for all benchmarks was less than two seconds and the maximum size of the checkpoint files was 95 MB for poppler. For all applications except indent the variance between checkpoint times was minimal. Since the indent infinite loop continuously allocates memory, taking a checkpoint later in the execution of this program records more program state, which requires more time and produces a larger checkpoint file.

4.5 Escape Results

Table 5 presents the summary of the results of the continued execution of benchmarks after using each escape strategy. In addition to this summary, we present detailed descriptions of escaping the loops in php, wireshark, gawk, apache, and pam in Section 5. We previously presented detailed descriptions of escaping the infinite loops in ctags, grep, ping, look, and indent in the evaluation of Jolt [24].

Continued Execution Errors. Columns 2 and 3 of Table 5 present errors in continued execution after applying each of the two escape strategies. The errors include unexpected application terminations, latent memory errors, and unresponsiveness. Entries marked as "None" did not exhibit any error as a consequence of infinite loop escape.

For 14 out of the 15 benchmarks at least one escape strategy enabled the application to execute without error in the continued execution. Instructing Bolt to escape the loop in java-vm with both the Unwind and Break escape strategies kept the application in an unresponsive state.

Seven benchmarks had an error after applying one of the escape strategies. Two benchmarks (php and wireshark) reported errors in parsing parts of their inputs after exiting the infinite loops. Two applications (gawk and ctags-fortran) experience segmentation faults after escaping the infinite loop with the Break strategy. Applying the Break strategy on pam, poppler, and grep-match leaves the applications unresponsive even after repeated escape attempts. A dagger (†) on an error entry indicates that the error triggered the application's existing error handling code, which allowed the application to terminate gracefully.

Termination Comparison. Columns 4 and 5 of Table 5 present the results of comparing the outputs obtained after terminating the application with the outputs produced by applying Bolt to escape the loop. For entries marked "Better" Bolt helped the application produce more output or handle more requests compared to termination. For entries marked "Same" both Bolt and termination produced the same output. If the escape strategy left the application in an unresponsive state, we mark the effect of escape as "N/A". We marked the entry for apache and the Break escape strategy as "Worse" because for some configurations of the server the continued execution may grant unauthorized access to protected data (see Section 5.4).

For 13 loops at least one of Bolt's escape strategies provided better output than terminating the application. For the remaining loop (look) termination and Bolt both produce the same output because the infinite loop occurs at the very end of the computation (this output is also the same as the output of the correct program).

Manual Fix Comparison. Columns 6 and 7 of Table 5 present the results of comparing the outputs produced by ap-

plying Bolt to the outputs produced by a later version of the application in which developers fixed the infinite loop. Entries marked "Same" indicate that the outputs from applying Bolt are identical to the outputs from the fixed application. Entries marked "Partial" produced some (but not all) of the correct output. Entries marked "None" produced none of the output that the manually fixed application produced. For entries marked with an asterisk (*), the output after escaping from the loop contained the complete or partial expected output from the fixed version, but also included additional output from code that is not executed in the fixed version.

For all loops except the java-vm, at least one escape strategy enabled the application to produce a partial output. For 7 loops, one of the escape strategies provided output identical to that of the manually fixed application. For the remaining 7 loops Bolt helped the application produce a partial result.

Four benchmarks that produced additional outputs after applying Bolt are wireshark, grep-match, ping, and indent. In the case of the Break strategy for wireshark, the application prints an additional warning message. In the case of the Unwind strategy for grep-match, the application outputs additional end-of-line characters. In the case of the Break strategy for ping, the extra output results in displaying the time stamp information of an input packet; the fixed version of the program ignores this part of the packet. In the case of the Break strategy for indent, the output (a C source file) is semantically identical—it differs by only a single additional whitespace character at the end of the file.

Comparison of Two Escape Strategies. Column 8 of Table 5 presents which one of the two escape strategies, Break or Unwind, was more effective for each infinite loop. We evaluated the relative effectiveness of the strategies by comparing the output produced after applying each strategy to the output produced by the manually fixed application. The entries for which both escape strategies produced identical outputs are marked as "Same".

For 4 loops, the Break strategy gave the best output, and for 7 loops, the Unwind strategy gave the best output. In the remaining 3 cases both the Break and the Unwind produced the same output. Both escape strategies were unsuccessful for the java-vm loop.

4.6 Infinite Loop Patterns

Out of the thirteen infinite loops, eight loops perform string pattern matching (look, apache, indent, and all ctags and grep loops), two loops traverse complex data structures (wireshark and ping), two loops perform numerical computation (php and java-vm), and one loop performs string substitution (pam). Within this diversity of computations, we identified two main reasons for the infinite loops in our benchmark applications:

- **Missing Transition:** While processing its input, the computation inside a loop enters a state from which it cannot proceed and consume the remaining input. Examples include unhandled zero-length string matches (the three grep loops), unhandled nested character patterns in a string (ctags-python), unmatched keywords in the input source code file (ctags-fortran), unsupported optional parts of the input message (ping), or undetected round-off errors in arithmetic operations involving subnormal floating-point numbers (php and java-vm).
- **Missing Exit Condition:** These loops have completed processing their inputs, but the exit condition is not satisfied. Examples include missing end-of-line character checks (indent and look), missing return statements in error-handling code (wireshark and pam), or missing string length check (apache).

Each of these patterns also correlates with how well escaping the infinite loop emulates the developers' manual fix (Table 5, Columns 6 and 7). For loops with missing transitions, escaping the loop produces a partial result in 5 out of 8 cases—the remaining two loops (php and ping) produce the same result as the manual fix and java-vm does not produce any result. The intuition behind this result is that the application enters an infinitely looping state before processing the entire input and, therefore, escaping such a loop curtails the execution of the loop without processing the remainder of the input.

For all loops with a missing exit condition at least one of Bolt's escape strategies produces the same result as the manual fix proposed by the developers. This is again intuitive because the application will have already finished processing the input by the time it enters a repeating sequence of states.

5. Detailed Case Studies

We next provide a detailed analysis of Bolt for five benchmark applications. These applications include both client applications and libraries (Wireshark, Gawk, and the PAM libraries) and server-side applications (Apache and PHP).

5.1 PHP

We continue with the example infinite loop from Section 2. PHP version 5.3.4 for 32-bit x86 processors when compiled with the gcc compiler (which is the default compiler for the binary distribution of PHP) contains an infinite loop when converting certain strings into floating point values [17].

The error occurs in the `zend_strtod` function in the file `zend_strtod.c`. The loop begins on line 2313. This function takes as input a pointer to a string literal (which can be a constant or an input that a user interactively provides as the script executes) and returns 1) the floating-point value of the largest convertible prefix of the string and 2) a pointer to the end of the prefix. We presented an input that triggers this infinite loop in Figure 1 in Section 2.

The infinite loop occurs when the loop continuously tries to adjust the accuracy of the floating point result as it reads

additional characters from the string. The computation does not make progress due to the interplay between the gcc compiler and the CPU's floating point unit—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 precision floating point value. When the computation tries to add the adjustment to the current result, due to rounding it leaves the value of the result unchanged so that it misses the loop's exit condition.

Infinite Loop Detection. Each iteration of the loop accesses over 20 local variables and executes several levels of nested function calls. A closer examination of the computation reveals that 1) in each iteration of the loop the computation allocates and frees temporary heap data structures that represent unbounded integers and 2) the only state in the loop that changes between iterations is the pointers to these data structures, which are independent of the loop's exit condition. Because PHP uses a custom bounded-size allocator for these data structures, the allocator eventually reuses the same (previously allocated and freed) memory locations after four loop iterations. This loop therefore repeats the same execution pattern every four iterations.

Effects of Escaping Infinite Loop. When using the Unwind escape strategy, PHP terminates and prints an error pointing to the user's source code: `Parse error: syntax error, unexpected $undefined in test.php on line 2.` Line 2 in the file `test.php` contains the floating-point string literal. The error handling code is triggered by intermediate values that are present in the registers at the time of applying the Unwind escape strategy.

When using the Break escape strategy, PHP continues its execution from an instruction still inside the loop that is in a branch not taken by the original computation, but which changes the values of the adjustment variables. This branch does not change any of the digits of the resulting floating-point 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. As a result, the user's script is properly parsed and executed, printing out a double value, without any visible errors.

For the Break escape strategy, we used Valgrind to check for latent memory errors. Valgrind does not report any memory errors and a manual inspection of the application's code shows that the branch at which Bolt forces the application to continue its execution ensures that the currently allocated memory is properly freed.

Comparison with Termination. If the floating-point value is passed as a constant (as in the example), then the infinite loop occurs during PHP's compilation phase and termination results in no PHP code being executed (PHP parses the entire file before executing the code).

Although applying the Unwind escape strategy also causes the code to not be executed, it still provides a parse error that points to the line where the value appears in the user's code. The Break strategy, on the other hand, enables the program to terminate and fully execute every instruction of the code, including the instruction with the string literal.

The infinite loop can also be triggered when reading the string literal from an input source, such as a file or an HTTP request. In this case, termination prevents any code past this point from being executed. The Unwind escape strategy will terminate the script (but will log the location of the error). And the Break strategy will allow the script to continue its execution past this point.

We note that while PHP has the facility to detect and terminate long-running scripts (specifically by setting a limit on the maximum execution time), PHP cannot recognize and terminate a script that contains this infinite loop because PHP checks to see if the limit has been exceeded only after executing (interpreting) complete PHP instructions. However, this infinite loop occurs in the middle of interpreting a single PHP instruction, so PHP is unable to detect that the script has exceeded its maximum time limit.

Comparison with Manual Fix. The developers manually fixed the application by marking local variables as volatile, which forces the compiler to store the intermediate values of the computation in memory (thus implicitly converting these values from 80-bit to 64-bit values). This avoids the undesired rounding of these variables.

We compared the result of applying the Break escape strategy to the faulty version of PHP with the result of the manually corrected version of PHP (version 5.3.5). The results of both executions had all digits identical.

5.2 Wireshark

Wireshark is commonly used for network traffic monitoring, recording, and analysis. It provides a graphical interface that enables a user to analyze network traffic in real time. Version 1.4.1 of Wireshark contains an infinite loop in the ZigBee wireless protocol module [14]. The infinite loop can be triggered by reading a malformed packet log file or by a remote user maliciously sending corrupt packets.

Figure 4 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 loop termination condition will never be satisfied. This loop is located in the function `dissect_zcl_discover_attr_resp()`. The loop begins on line 1192 of the file `packet-zbee-zcl.c`.

```
while ( *offset < tvb_len
      && i < ZBEE_ZCL_NUM_ATTR_ETT ) {
  if( tree ) {
    // ...
    i++;
    // ...
  }
}
```

Figure 4: Wireshark Loop in `packet-zbee-zcl.c`

Infinite Loop Detection. When gcc compiles the application with optimizations turned on (with optimization level `-O2`), the compiler executes a loop unswitching optimization, which splits the original loop into two loops representing individual branches of the `if` condition, and moves the `if (tree)` condition outside to control which loop to execute. If the `tree` pointer is not `NULL`, then the program executes the loop that processes `tree` and increments the induction variable `i`. If the pointer `tree` is `NULL`, then execution continues in the one-instruction long infinite loop in Figure 5.

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

Figure 5: Disassembly of the Wireshark Infinite Loop

While this infinite loop is the simplest of all loops in our benchmarks, it deserves attention because Wireshark is a multithreaded application (see Section 3.1.2). In this case, the infinite loop does not make any memory accesses and, therefore, Bolt can be certain that a computation in another thread will never cause this loop to exit.

Effects of Escaping Infinite Loop. With both the Unwind and Break strategies, Wireshark becomes responsive after Bolt escapes the infinite loop: it presents the network traffic that caused the infinite loop and all subsequent traffic. The function that contains the infinite loop does not contain any code after the loop, so applying the Unwind strategy does not skip any computation.

The Break strategy transfers the execution of the program to the code within the `if` branch in Figure 4 (since the compiler places this code below the infinite loop in the program's binary); this forces Wireshark to attempt to parse a corrupted packet. However, every operation within the body of this `if` branch has a redundant check to ensure that `tree` is not `NULL`. Therefore, if `tree` is `NULL` (as in the this case), execution simply exits the loop. As a result, the computation returns a status message noting that the packet is malformed.

Comparison with Termination. Terminating the program results in lost log data as the infinite loop prevents the user from saving the log data before termination. In contrast, both of Bolt's escape strategies enable the application to become responsive again and continue analyzing traffic, including the packet that caused the infinite loop.

Comparison with Manual Fix. The manual fix by the developers simply moved the check for `NULL tree` pointer values 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 and 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 an additional `[Malformed Packet]` status message.

loop.awk:

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

input:

```
''Italics with an apostrophe'' embedded''
''No nested apostrophes''
```

Figure 6: Script and Input that Cause the Long-Running Loop in Gawk

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 a long-running loop in its regular expression library [11].

Figure 6 shows the awk program and the input file (derived from the bug report) that cause gawk to become unresponsive. 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>` (emphasis) tags. The result of the substitution on each line is then printed out. In this case, the loop is triggered when gawk encounters a string with nested double quotes.

Long-Running Loop Detection. The long-running loop is located 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 [11]:

```
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.
```

Bolt identifies that the state after each loop iteration changes and does not recognize this loop as infinite. Testing gawk with smaller examples that we constructed by removing the letters from the beginning of the input from Figure 6 reveals that the loop is not infinite, but its execution time doubles with any character added before the nested quote symbol. The execution of gawk for the input from Figure 6 terminates after about one hour of execution.

Further examination of the loop shows that gawk attempts matching the part of the string up to and including the nested quote symbols. However, because the substring does not fully match the pattern (i.e., there are additional characters after the quote symbols), gawk backtracks the search and tries to match all possible prefixes, often repeating some of them multiple times. The number of prefixes gawk tries is exponential in the size of the number of characters before the quote symbols.

Effects of Escaping Long-Running Loop. If Unwind sets the return value to 0 or uses the value residing in the return register from the function that contains the loop, the program terminates with a memory error. Gawk contains a signal handler that captures this type of error and prints the er-

ror message `gawk: loop.awk:1: (FILENAME=input FNR=1) fatal err: internal error`.

If Unwind instead returns -1 from the function that contains the loop, the program will continue to execute and will produce the following output for the example input file:

```
''Italics with an apostrophe<em>'' embedded''</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. Valgrind detected no memory leaks when using this escape strategy. A manual inspection of the loop shows that the computation uses the value -1 as an error code representing that no match was found.

After escaping from the loop with the Break strategy, gawk terminates and prints the same error message as in the case when Bolt applies the Unwind strategy with return value 0.

Comparison with Termination. Terminating the program prevents gawk from processing any lines in input files after the line that elicits the long-running loop. The Unwind strategy allows gawk to process the remainder of input, while only affecting the result on the lines that caused the long execution of the loop.

Comparison with Manual Fix. The developers fixed this long-running loop as a part of a complete rewrite of awk's pattern matching engine for version 3.1.2 (in the bug report response that we cited, the developer announced the new version of the pattern matching library, which at the time was in development). We used this version of gawk to process this input file and it instantaneously produced the following result :

```
<em>''Italics with an apostrophe</em>'' embedded''
<em>''No nested apostrophes''</em>
```

In general, regular expressions match from left to right and, therefore, the manually fixed pattern matcher inserts tags around the first matched quotes. The Unwind strategy instead finds the second match on this line. However, it produces legal HTML tag pairs, which do not affect rendering of the remainder of the output file. The lines without nested quotes in the output are identical in both cases.

5.4 Apache

Apache HTTP server version 2.2.18 contains an infinite loop in string matching code in the Apache Portable Runtime Library, version 1.4.4 [15]. The error occurs in the loop starting on line 199 in the function `apr_fnmatch`, in the file `apr_fnmatch.c`. This function takes as inputs a string, a pattern to match, and a set of flags that controls properties of the matching. It returns 0 if a match was found and 1 if no match was found.

Access Policy:	**Request:**

`<Location "/*/WEB-INF">`
`deny from all`
`</Location>`
	`GET /test HTTP/1.1`
	`Host: 127.0.0.1`

Figure 7: Access Policy and HTTP Request that Cause the Infinite Loop in Apache

Apache uses the `apr_fnmatch` function as a part of computation that, given a set of access policies, determines whether to allow or deny requests that the server receives. Figure 7 presents a sample access policy and HTTP request that causes the infinite loop. This access policy is intended to prevent clients from accessing directories matching the regular expression `"/*/WEB-INF"`. Apache treats slash characters ('/') is a special way in this expression: a string that matches this pattern must have exactly two slash characters. The HTTP request contains the path `/test`, which does not match the pattern of the policy. The infinite loop occurs when the non-matching requested path has less slash characters than the pattern.

Apache calls the function `apr_fnmatch` with three arguments: the requested path (`"/test"`), the access policy pattern (`"/*/WEB-INF"`), and a flag to ensure the number of slash characters in both the path and the pattern are the same. After the first iteration, the infinite loop consumes the entire requested path, but only partially consumes the access policy pattern (the corresponding pointer is incremented to point to the second slash-character). The following iterations of the loop, however, do not recognize that the requested path is consumed and try (unsuccessfully) to match the remainder of the access policy pattern with the empty string.

Infinite Loop Detection. The loop does not change the state of the application between iterations and the loop only makes calls to the string library routines. This infinite loop occurs in the Apache Runtime Library code, which is dynamically loaded at runtime (Bolt readily supports analyzing code inside shared libraries).

Effects of Escaping Infinite Loop. Apache interprets the effect of the Unwind strategy with a return value other than 0 as returning a no-match—the requested path does not match the access control policy pattern and the server should not apply the policy. When Apache continues execution after the loop, it makes recursive calls to the function `ap_process_request_internal` and, for the same request, the function `apr_fnmatch` and the infinite loop will execute again, in a different stack frame. We can again apply the Unwind strategy using a non-zero return value and Apache will continue the execution and send a response to the client.

Using the Break strategy or the Unwind strategy with a zero return value allows the server to continue execution with a return value 0 of the function, indicating that the path

matches the pattern. This return value indicates that Apache should apply the access control policy. The response of the server depends on the policy's rule. In the previous example, the *deny all* policy instructs the server to deny access to the path and instead send a "`403 Forbidden`" response.

A manual inspection of the infinite loop indicates that no memory is allocated or freed during the loop execution—the loop memory accesses include only pointer arithmetic and comparisons of local variables.

Comparison with Termination. While the server is in the infinite loop, a client will keep waiting for the response until a timeout on the client side or until the server process is terminated. Termination of the server process also terminates the connection and as a consequence the client will not receive any reply data.

The Unwind strategy with non-zero return value instructs Apache not to apply the access control policy. This is a correct behavior, as the pattern and the path does not match the policy. The server eventually sends the correct response to the client's request.

The Break strategy and the Unwind strategy with zero return value instruct the server to apply the access control policy. As a result, depending on the body of the policy, the server can send the correct response, send the permission error instead of requested content (as for the input from Figure 7), or allow this policy to override the previously applied policies (which may potentially lead to security issues by allowing unauthorized access to protected paths on the server).

Comparison with Manual Fix. The developer's fix, in APR version 1.4.5 (Apache 2.2.19) added two checks to this computation. The first check at the start of the loop body ensures that the computation exits the loop if the input string has been completely consumed. The second check after the loop body ensures that the function `apr_fnmatch` returns 0 (match found) only when both the requested path and the policy pattern have been completely consumed. Otherwise the function returns a non-zero value (no match found). For the inputs from Figure 7, the fixed application will exit the loop once the requested path is consumed and then return a non-zero value (no match). This fix produces the same result as our Unwind escape strategy with non-zero return value.

5.5 Pluggable Authentication Modules (PAM)

Pluggable Authentication Modules (PAM) is a shared library that implements authentication mechanisms. Many Linux programs and utilities, for example login, su, and sshd, use PAM for authentication. PAM version 1.1.2 contains an infinite loop that can be triggered by a long environment variable expansion [16]. The function that performs this expansion is `_expand_arg` in the `pam_env` module. The loop begins on line 553 of the file `pam_env.c`.

Because PAM modules are not standalone, we discuss the loop in the context of the the `su` tool and the command `'su - $USER'`, which allows the current user to login with another

```
EF_255 DEFAULT=BBBBBBBB... [repeated 255 times]
EF_256 DEFAULT=${EF_255}B
EF_1024 DEFAULT=${EF_256}${EF_256}\
                ${EF_256}${EF_256}
EF_8191 DEFAULT=${EF_1024}${EF_1024}\
                ${EF_1024}${EF_1024}\
                ${EF_1024}${EF_1024}\
                ${EF_1024}${EF_256}\
                ${EF_256}${EF_256}\
                ${EF_255}
EVIL_OVERFLOW_DOS DEFAULT=${EF_8191}AAAA
```

Figure 8: Entries in `.pam_environment` File that Cause the Infinite Loop in PAM

```
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);
  }
// ...
}
```

Figure 9: PAM Infinite Loop Code in `pam_env.c`

user name `$USER`. The tool first prompts the user for the password, which will be authenticated using PAM. After authentication, PAM creates an execution shell and sets up the environment variables defined in the `~/.pam_environment` file. The `pam_env` module reads this file and calls the `_expand_arg` function. This function contains a loop that scans the input string for variables to expand.

Figure 8 presents an input file for the `.pam_environment` file that triggers the infinite loop. Each entry is a single environment variable, followed by the keyword `DEFAULT` and the default value. The value of the environment variable `EVIL_OVERFLOW_DOS` is constructed by concatenating the values of the previously defined environment variables. Its length exceeds 8192 characters.

Figure 9 presents the simplified version of the loop that expands and assigns values to environment variables. PAM limits the length of the value of each environment variable to 8192 characters (this is the size of an internal buffer used to store the values). During every expansion, the current length of the value is compared to the maximum size of the internal buffer. If this length is greater than the maximum size (as in the case of the `EVIL_OVERFLOW_DOS` variable), the computation uses the `pam_syslog` function to log this event. However, the computation does not exit the loop and, instead, keeps logging the same event in every subsequent iteration.

Infinite Loop Detection. This loop iterates through a large number of states before repeating. The reason for the multiple states is the allocation and deallocation of temporary buffers that PAM's logging module creates every time it prints data to the output file in the function `pam_syslog`.

Effects of Escaping Infinite Loop. The Break strategy does not escape this infinite loop. It instead continues the execution inside the same loop along a different path, previously unseen by the Bolt Detector. The computation in this execution path interprets one character 'A' from the input as an unrecognized escape character. On subsequent loop iterations, execution returns to the original path and the execution remains in the infinite loop.

The Unwind strategy with non-zero values allows the execution to escape the loop. The Unwind return value is compared with zero (which represents success) and the caller function returns an error code up the call stack. This error code is handled by the clients of the PAM module. PAM then continues the computation and invokes additional modules that again enter the same infinite loop. Escaping the loop for the second time returns the error code to the client. Each client may use a different strategy to handle this error code: `su` prints an error message `'su: Critical error - immediate abort'` and terminates. Other clients, such as a graphical login screen, may refuse to log in the current user, but still enable logging in for other users.

The Unwind strategy with a value of zero (applied two times, as in the previous case) also escapes the infinite loop and allows the program to continue executing. A return value of zero indicates the call to the function has been successful and all variables expanded correctly, allowing `su` to create and start a new shell process. However, after escaping the loop with Bolt, the new shell remains in the background and becomes unresponsive (the shell expects to run in the foreground and have a direct access to the terminal). While the current Bolt implementation leaves the new shell in an unusable state, using gdb to manually simulate this escape strategy does allow the shell to execute in the foreground and a user to interact with it. The environment in this shell contains all the variables declared in the `.pam_environment` file, including `EVIL_OVERFLOW_DOS`, which has the same value as `"${EF_8191}"`.

We note that this infinite loop does not cause privilege escalation. The PAM module that performs authentication part has already granted permissions to the user by the time the application reaches the module with the infinite loop. Manual inspection shows that no memory is allocated in this function until the very end, but the program never executes this code after applying the Unwind strategy.

Comparison with Termination. Terminating the process when using `su` has the same effective result as using the Unwind escape strategy with non-zero return value. We note that other clients that use the PAM libraries may handle an internal PAM error in different ways. Some clients may terminate as a result of the error code returned by the authentication module (as in the case of su), but in other cases, the client may print an error message and continue interacting with a user on other tasks.

Comparison with Manual Fix. We compared the Bolt execution with the manually fixed PAM libraries version 1.1.3. The result of invoking the `su` command with the fixed version was identical to the Unwind strategy with the non-zero return value: `su` prints an error message to the screen and terminates the execution.

A manual inspection of the code reveals that the function `_expand_arg` is private to the PAM module, and all calls to this function are wrapped by the internal error handler that returns to the clients of the module one of the defined error codes. Applying the Unwind strategy and running the manually fixed version of PAM return the same error code to clients.

Using the Unwind escape strategy with value 0 presents an alternative fix for this computation—the environment variable that causes the infinite loop can be truncated or discarded, and PAM can log a warning message and still start the new shell.

6. Related Work

Infinite Loop Detection. Researchers have developed several techniques that use symbolic program execution to detect infinite loops [22, 28]. Unlike Bolt, these techniques require the source code of an application and do not provide the user with the option to escape an infinite loop and continue the application's execution. Researchers have also developed techniques for program debugging and verification that ensure that a program does not contain any infinite loops [19, 20, 25, 26, 49].

Jolt. Our previous work on Jolt [24] was the first to show that a simple, lightweight technique for dynamically detecting and exiting infinite loops at runtime could enable applications with infinite loops to produce acceptable outputs. Jolt used a compiler to insert instrumentation into the program. It therefore required access to source code, a build environment, and ahead of time planning for infinite loop detection and escape before executing the application. The inserted Jolt instrumentation also imposed a 2%-10% overhead in standard production use. Bolt, in contrast, is fully dynamic and operates directly on stripped binaries with no need for recompilation, no access to source, an on-demand usage model, and no overhead in production use. Bolt also handles multithreaded applications and can detect infinite loops in which the loop state repeats only after multiple iterations of the loop. Because Bolt operates on binaries, it can detect and escape infinite loops that appear in any part of the application, including loops that appear in external libraries, which may not be available for recompilation.

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 [38]. The proposed technique bounds the number of iterations of all loops in a program. The bounds are determined empirically by observ-

ing the execution of the program on representative inputs. In comparison to Bolt, Nguyen and Rinard's technique is completely automated, but may also escape loops that would otherwise terminate.

Like Bolt, ClearView [39] is designed to repair errors in potentially stripped binaries. ClearView, however, targets a different class of errors—specifically, errors that violate learned invariants. ClearView is also designed to operate without user interaction—it relies on a set of pluggable error detectors to determine when the application is executing incorrectly. Bolt, in contrast, relies on the user to determine when the application has become unresponsive (although it would be straightforward to extend Bolt to trigger its loop detection and escape mechanisms based on a timeout). Finally, ClearView uses a community approach—it learns from past experience to favor repairs that have succeeded in the past.

Failure-Oblivious Computing [43] enables applications to dynamically identify and recover from out-of-bounds memory reads and writes. 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, the currently implemented version of Failure-Oblivious Computing uses a compiler to insert checks and therefore requires access to the source code of the application. Like Bolt, Failure-Oblivious computation enables applications to survive otherwise fatal errors. The two techniques are potentially synergistic in that they may, together, enable applications to survive combinations of errors that either technique would be unable to handle separately.

The Rx system [40] takes periodic checkpoints of the program state. When a failure occurs, it 1) reverts the application to the checkpoint, 2) makes semantically equivalent changes to the application's execution environment (e.g., adding padding to allocated buffers), and 3) restarts the execution. Not that Rx does not attempt to change the application's semantics—it instead attempts to search the set of existing executions to find one that does not have an error. Like Rx, Bolt combines checkpoints with recovery actions. But one critical difference between the two systems is that escaping infinite loops changes (purposefully) the semantics of the original application to eliminate a bug. Such an approach is inherently required to eliminate infinite loops (or, for that matter, many other software errors).

Researchers have also proposed a number of techniques that automatically repair applications by statically manipulating their code (or scheduler in the case of deadlocks) [27, 30, 31, 33, 39, 46, 47, 50, 51]. These systems operate in an off-line fashion: they first observe the program failure, and only after do they generate repairs for future executions. Therefore, these systems cannot be used to recover an output from a failed application. Bolt differs in that it focuses exclusively on infinite loops and allows applications to recover even the first time they encounter the error.

Loop Perforation and Task Skipping. Infinite and long-running loop escape can be viewed as a form of loop perforation [36, 37, 48, 53], which reduces the execution time or energy consumption of an application by skipping iterations of time-consuming loops. Task skipping obtains similar benefits by skipping tasks in time-consuming computations [44, 45]. Like loop perforation and task skipping, the goal of loop escape is to enable the application to produce an acceptable result in an acceptable time frame (although the reduction in execution time is larger for infinite loop escape than for loop perforation). Both loop perforation and infinite loop escape are forms of acceptability-oriented computing [23, 42] in that they are both mechanisms that enable applications to deliver acceptable results within an acceptable time frame.

Unsynchronized Parallel Computing. Unsynchronized updates to shared data can often be coded so that the net result of any resulting data races is simply dropping one or more updates (which ensures that the updated data structures still satisfy key data structure consistency properties) [41]. It is possible to apply this principle to obtain parallelizing compilers that produce parallel programs with data races [35]. Despite the presence of these data races, the automatically generated parallel code produces accurate enough results often enough to be acceptable. Like infinite loop termination, these techniques may improve a program by eliminating computation.

7. Conclusion

Bolt enables users to escape infinite and long-running loops in unresponsive applications so that the application can produce useful output or continue on to successfully process additional inputs. Bolt operates directly on stripped x86 and x64 binaries. It requires no access to source code, no recompilation, imposes no overhead in standard production use, works with multithreaded applications, implements multiple loop escape strategies, and supports an on demand usage model in which it attaches and detaches to and from running applications. Our experimental results show that Bolt can effectively enable applications to escape infinite and long-running loops, produce useful output (in many cases the same output as subsequent versions with the infinite loops eliminated through developer fixes), and continue on to process additional inputs and serve the needs of its users.

Acknowledgements

We would like the thank Deokhwan Kim, Stelios Sidiroglou, and the anonymous reviewers for their useful feedback and comments. We note our previous work on this topic [32]. This research was supported in part by the National Science Foundation (Grants CCF-0811397, CCF-0905244, CCF-1036241, and IIS-0835652), DARPA (Grants FA8650-11-C-7192 and FA8750-12-2-0110), and the United States Department of Energy (Grant DE-SC0005288).

References

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

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

[3] BLCR frequently asked questions. https://upc-bugs.lbl.gov/blcr/doc/html/FAQ.html# porting.

[4] Common Vulnerabilities and Exposures (CVE). http://cve.mitre.org/.

[5] GNU awk. http://www.gnu.org/s/gawk/.

[6] The libunwind project. http://www.nongnu.org/libunwind/.

[7] Linux PAM Modules. https://fedorahosted.org/linux-pam/.

[8] PHP. http://www.php.net/.

[9] Poppler. http://poppler.freedesktop.org/.

[10] Wireshark. http://www.wireshark.org/.

[11] Gawk: Infinite loop in sub/gsub. http://lists.gnu.org/archive/html/bug-gnu-utils/2002-10/msg00051.html, 2002.

[12] Java VM: Fix bug 4421494 infinite loop while parsing double literal. http://bugs.openjdk.java.net/show_bug.cgi?id=100119, 2009.

[13] Poppler: Problem decoding JBIG2Stream stream. https://bugs.freedesktop.org/show_bug.cgi?id=23025, 2009.

[14] Wireshark: Bug 5303 - Infinite Loop in ZCL Discover Attributes dissection. http://bugs.wireshark.org/bugzilla/show_bug.cgi?id=5303, 2010.

[15] Apache: apr_fnmatch infinite loop on pattern "/*/WEB-INF". http://issues.apache.org/bugzilla/show_bug.cgi?id=51219, 2011.

[16] PAM: 100% CPU utilization in pam_env parsing. http://bugs.launchpad.net/ubuntu/+source/pam/+bug/874565, 2011.

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

[18] Working Draft, Standard for Programming Language C++. www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf, 2011.

[19] E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination analysis of Java bytecode. In *FMOODS*, 2008.

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

[21] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, 2004.

[22] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen. Looper: Lightweight detection of infinite loops at runtime. In *ASE,* 2009.

[23] M. Carbin, D. Kim, S. Misailovic, and M. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.

[24] M. Carbin, S. Misailovic, M. Kling, and M. Rinard. Detecting and escaping infinite loops with Jolt. In *ECOOP*, 2011.

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

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

[27] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE*, 2009.

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

[29] J. Hollingsworth, B. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *SHPCC*, 1994.

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

[31] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *OSDI*, 2008.

[32] M. Kling. Escaping infinite loops using Bolt. MEng. Thesis, MIT CSAIL, January 2012.

[33] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, 2012.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[35] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *Transactions on Embedded Computing Systems (to appear)*, 2012.

[36] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *SAS*, 2011.

[37] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. C. Rinard. Quality of service profiling. In *ICSE*, 2010.

[38] H. Nguyen and M. Rinard. Detecting and eliminating memory leaks using cyclic memory allocation. In *ISMM*, 2007.

[39] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP,* 2009.

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

[41] M. Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm. Technical Report MIT-CSAIL-TR-2012-005, MIT, Feb. 20012.

[42] M. Rinard. Acceptability-oriented computing. In *OOPSLA Onwards*, 2003.

[43] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and W. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, 2004.

[44] M. C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, 2006.

[45] M. C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA*, 2007.

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

[47] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX Technical*, 2005.

[48] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.

[49] F. Spoto, F. Mesnard, and E. Payet. A termination analyzer for Java bytecode based on path-length. *Transactions on Programming Languages and Systems*, 32:1–70, March 2010.

[50] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[51] J. Wu, H. Cui, and J. Yang. Bypassing races in live applications with execution filters. In *OSDI*, 2010.

[52] V. C. Zandy, B. P. Miller, and M. Livny. Process Hijacking. In *HPDC*, 1999.

[53] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, 2012.