Hybrid Analysis Tools for Computer Systems Education

Eric Williamson

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science & Application

Godmar Back, Chair
Ali Butt
Dongyoon Lee

May 9, 2018
Blacksburg, Virginia

Keywords: CS Education, Program Analysis, Tools

Hybrid Analysis Tools for Computer Systems Education

Eric Williamson

(ABSTRACT)

To learn about computer operating systems, students at Virginia Tech implement a command-line shell in their Computer Systems course. Successfully implementing the shell requires a deep understanding of operating system abstractions and interactions. Students often struggle with the project because subtle errors can take hours to debug.

In this work, we developed two hybrid domain-specific analysis tools to pinpoint the root causes of student errors: EshMD and ShellTrace. The EshMD tool models common errors in the shell and checks the student code against those models. To accomplish this, it monitors the specific calls the program is making and correlates those with expected changes in its environment. Students' errors are shown directly in the source code. The concept of EshMD can be applied to other programming projects by observing and modeling common bugs during implementation.

The ShellTrace tool dynamically creates a specification from a reference solution based on how the reference solution makes use of operating system resources and then uses this specification to check that a student solution is functionally identical. The ShellTrace concept can be applied to other programs that exhibit similar resource dependencies.

We deployed these tools in an undergraduate computer systems class and evaluated our tools based on the number of bugs detected and the students' perceptions of usefulness. We found that the tools detected a significant number of bugs and that the majority of students that made use of the tools found them valuable in debugging their submissions.

Hybrid Analysis Tools for Computer Systems Education

Eric Williamson

(GENERAL AUDIENCE ABSTRACT)

To learn about computer operating systems, students at Virginia Tech implement a command-line shell in their junior-level Computer Systems course. A command-line shell is a computer program that allows the user of a computer to run other programs by typing in the program names. The command-line shell will then run those programs on the user's behalf. The command-line shell project requires students to understand and use the abstractions that the underlying operating system provides to a computer program. Students often struggle with the project because subtle errors in their implementation can take hours to address.

In this work, we developed two analysis tools to pinpoint the root cause of student errors: EshMD and ShellTrace. The EshMD tool models common errors in the command-line shell and checks the student code against those models. Students' errors are shown by directly pointing to the error locations in their code.

The ShellTrace tool dynamically creates a project specification from a solution written by the course staff and checks that a student solution meets the specification. Generating from the staff solution allows checking the student solution without having to encode the specific project requirements into the tool itself.

We deployed these tools in an undergraduate computer systems class and evaluated our tools based on the number of errors detected and the students' perceptions of usefulness. We found that the tools detected a significant number of errors and that the majority of students that made use of the tools found them valuable in finding and fixing the errors in their submissions.

Acknowledgments

I first would like to thank my advisor Dr. Godmar Back. He both encouraged and inspired me to strive for more than the adequate and sparked my interest in computer systems. Without first having him as a professor I would not have made the jump to grad school. He has dedicated countless hours to my success.

I would like to thank my committee members Dr. Dongyoon Lee and Dr. Ali Butt for the suggestions they provided to explain this work better and for their advice on future directions for this project.

I would also like to thank Christy Coghlan for giving numerous editing passes to my thesis and for forcing me to tackle it bit-by-bit.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The operating system (OS) is the link between user programs and the hardware they make use of. An OS provides abstractions to represent and interact with the resources it controls. Understanding the OS is essential to writing robust and scalable code that takes full advantage of the underlying hardware. As part of their undergraduate education, students of Computer Science must understand operating system concepts related to the creation and management of programs running on the OS, communication between programs, and the concurrent execution of programs. In addition to learning concepts, students are expected to be familiar with how OS abstractions are presented via a system call API. For instance, in the Linux OS, running programs are represented as processes and their communication with each other and outside devices is managed through file descriptors.

To learn these concepts and their representation in existing operating systems, students in the computer systems course at Virginia Tech (CS 3214) are expected to implement a control program called a command line shell, which serves as a user interface to the machine. The shell takes user input to determine which programs to run and executes them on the user's behalf.

The shell project requires students to make use of Linux's interfaces to create processes and manage and inform the user of those processes' progress. Implementing this functionality requires an understanding of the lifecycle of processes as well as using interprocess communication mechanisms. Students are expected to implement their shell within an environment

in which processes may execute arbitrary user commands concurrently on multiple processors. After successfully implementing the shell students will have gained a deep, hands-on understanding of process management, interprocess communication, and process-level concurrency.

1.1 Motivation

The shell project provides several unique challenges for undergraduate students. In response to user input, the student's shell implementation must make an intricate sequence of system calls to create processes and manage their execution and communication. Although students had to write code in previous courses that made use of encapsulated libraries such as the standard libraries of a programming language, the system call API is the first API many students encounter in which making such calls affects both the state of external entities as well as the state and control flow of the calling processes themselves. To build a robust shell, students must reason about not only the shell process they implement but also the child processes created by the shell and the system objects with which they interact.

For instance, in Linux, a child process is created by cloning an existing parent process when the parent process calls the *fork* system call. Students must correctly identify the parent and the child process to properly set up the child to execute the program of interest. Cloning the current process can cause confusion for some students as both processes will return from *fork* and will continue the execution of the shell's code as separate processes before branching into their respective roles as parent and child process.

In the implementation of the shell, students must create and manage file descriptors to facilitate interprocess communication. Interprocess communication is used to create pipelines of processes in which the output of one process is passed to the input of the next process.

Improper management of these file descriptors can lead to difficult to debug errors such as child processes failing to terminate or output data being lost or not accessible.

Because the shell is expected to allow the user to interact with arbitrary programs in a concurrent environment, misunderstandings of the semantics of system calls or their interaction with the environment can lead to errors whose symptoms do not point directly to the root cause. In this work, we provide automated tools that detect and report these errors in student solutions. Informative error detection can help students understand the implications of their mistakes.

Automated tools have been shown to be successful in industry. Developers use both static tools such as linters and Coverity [6] as well as dynamic tools such as Google's Sanitizers [36, 38]. These tools are general purpose and geared toward professionals who have an understanding of the complex interactions their code can have with the operating system but do not capture the specific bug patterns present in the shell project. By developing domain specific tools that embed knowledge of common bug patterns we have observed, we can help students understand the underlying reasons for their code's failure and pinpoint the root cause.

1.2 Core Contributions

Our contributions lie in the two domain specific analysis tools we developed: EshMD and ShellTrace. These tools analyze the system calls and other calls and assignments performed by the shell process and the child processes spawned by the shell to ensure that the student solution produces the expected functionality. We identified common student errors and questions related to improper API usage and misunderstandings of operating system semantics through both anecdotal evidence and studying bugs in past course offerings. We built a

web-based interface and integrated it into the course website to display results of the tools, including the location of errors in the codebase and an explanation of why the exhibited behavior is considered an error.

After providing these tools to students, we evaluated their effectiveness during the course. We counted and examined the bugs each of the tools detected and the students' comments on the tools after the submission of their project. The results of the bug detection statistics and student feedback indicate that the analysis tools detected many common student bugs and that many students found the tools useful in their debugging process. Students also provided feedback on the parts of the tools they found confusing, which could be used to improve the tools in future deployments.

1.3 Roadmap

In Chapter 2, we discuss general analysis concepts and provide the necessary background to understand the types of bugs commonly seen in the shell project. Chapter 3 discusses the EshMD tool, the categories of bugs it detects, and how it is implemented. Chapter 4 similarly discusses the ShellTrace analysis tool and establishes its differences from EshMD. In Chapter 5, we evaluate the effectiveness of both of the tools when deployed in an undergraduate computer systems class. Chapter 6 gives perspective on related works and this works' place. Chapter 7 looks at possible routes for expanding this research and Chapter 8 concludes our work.

Chapter 2

Background

2.1 Program Analysis

This section outlines the types of analyses that can be performed on a program. We discuss two types of analysis: static analysis and dynamic analysis, both of which are used in EshMD and ShellTrace.

Program analysis is an automated way to assess the behavior of a program. It can be used to assess the reliability, performance, or presence of bugs in a program. We focus on analyses that detect the presence of bugs. The terms soundness and completeness are commonly used to describe the guarantees provided by an analysis. A sound analysis will never have a false negative, meaning that it will always report an error if there exists an error. A complete analysis will never have false positives, meaning that it will never report an error in the program that is not present. An analysis can be made trivially sound by always reporting an error, and trivially complete by never reporting an error, thus the challenge lies in being both sound and complete.

2.1.1 Dynamic Analysis

Dynamic analysis is a type of program analysis that is performed by executing a program and identifying and collecting statistics about events of interest. Dynamic analysis can be used

for monitoring the validity of program invariants, for profiling the performance of a program, for feedback-driven optimization of just-in-time compiled programs [3], or for detecting bugs such as the misuse of existing APIs.

Dynamic analysis tools can collect events of interest by executing the code in a virtual machine that monitors events, or by instrumenting code with added monitoring capabilities. The analysis can be performed during the execution of the program or at a later point based on a generated event trace. Our use of dynamic analysis instruments the code during the compilation phase in order to minimize the overhead relative to uninstrumented execution.

Because dynamic analysis considers only the specific set of executions during which the program is run, for bug finding purposes the program must be run on a broad set of representative inputs to increase the chance of finding invariant violations or API misuses. We make use of dynamic analysis both in EshMD and ShellTrace and use an extensive test suite provided to the students to give a variety of inputs to the students' shells.

2.1.2 Static Analysis

Static analysis is a program analysis technique that is performed without running the program. Instead, either the source code, intermediate code, or object code is analyzed. Static analysis is commonly used by the compiler to perform type checking as well as to optimize the generated code. Outside of the compiler, static analysis has been used for a variety of purposes, including bug finding and ensuring adherence to coding style guidelines and best practices. In this work, we use static analysis to reason about the program states that result from bugs in API usage that stem from students' misunderstandings of API semantics.

Static analysis can analyze execution paths even when these paths would be executed only in special cases. This approach allows rare bugs to be detected even when there is a low

probability of their occurrence during normal execution or if they require a specially crafted input to trigger.

One static analysis method is symbolic execution [7, 11, 28] which represents the program variables as symbolic values and the program expressions as symbolic expressions. A symbolic value represents the set of all possible values a variable can take on at a given point during the program's execution. Program expressions can be interpreted symbolically as well to compute the sets of values an expression can take on. When combined with boolean predicates, symbolic expressions constrain the symbolic values of variables. A *state* in symbolic execution represents the current set of symbolic values and the symbolic expressions that refer to them. The execution of the program is simulated by creating and updating symbolic expressions using the program's syntax to generate a new state. If a program encounters a conditional if-statement symbolic execution creates two states, one that "takes" the branch and the other that does not, with updated constraints for each.

Since each conditional branch creates two states, a series of n conditional branches could lead to 2^n possible states, resulting in an exponential number of states in a program. The resulting state explosion [9] can make it infeasible to explore all possible states. For this reason, implementations of symbolic execution typically limit the number of states explored, resulting in false negatives when states that correspond to bugs are left unexplored.

2.2 LLVM

LLVM is a compiler infrastructure project that provides a set of technologies that support compiling a variety of programming languages [29, 30, 31]. LLVM has been used in a variety of academic research projects as well as in industry. It provides an easy-to-reason-about intermediate representation of the code and a defined API that permits analysis tools to be

inserted into the compilation toolchain.

When a C program goes through the LLVM pipeline, it first is read in by a front-end such as Clang [1] that builds an internal representation of the code (called an abstract syntax tree) and produces an intermediate representation for the subsequent compilation phases. A number of LLVM back-end passes then read that intermediate representation, perform optimizations, and produce machine code to be run. LLVM controls the execution of these passes automatically.

For this work, we focus on two specific program representations used in the LLVM framework: the abstract syntax tree representation and the LLVM intermediate representation.

2.2.1 Abstract Syntax Tree

An abstract syntax tree (AST) is a tree structure that represents the syntactic structure of a program. Each node represents a construct in the source language such as a conditional or a loop statement. The AST can be traversed to simulate execution or to look at certain properties of the program. Compared to using source code directly, using the AST simplifies analysis because the AST does not contain unnecessary information about the program's concrete syntax, such as punctuation (i.e., semicolons at the end of statements or parentheses to group expressions) or delimiters (whitespace between operators).

Clang Static Analysis

Clang [1] provides a static analyzer that performs analysis on the AST to detect and report potential program bugs. It includes several built-in checkers that detect common generic bug patterns such as passing uninitialized values to functions and null pointer dereferences. We make extensive use of the Clang static analyzer framework in EshMD by adding checkers

that model system calls of interest to identify paths that match bug patterns for the shell project.

Clang's built-in static analyzer performs a path sensitive analysis [15] as it traverses a path in the AST, using the symbolic execution approach discussed in Section 2.1.2. This analysis traverses the nodes of the AST and computes the sets of possible values that each variable or expression can take on an individual path. Since these sets can be very large, they are represented as sets of ranges where possible. The analyzer's built-in modules model arithmetic operations and perform the proper constraint evaluation on conditional branches.

Our bug checkers interact with these existing modules to model the effects of specific function calls, such as by adding constraints associated with the return values of these functions (e.g. *fork()* may return -1, 0, or a number greater than 0 to signal failure, return in the child process, and return in the parent process, as will be discussed in Section 2.3.1). Consequently, no additional logic for propagating these values along subsequent analysis paths is required.

2.2.2 LLVM Intermediate Representation

The LLVM Intermediate Representation (IR) is a strongly typed programming language that serves multiple purposes. In addition to being the compiler's in-memory representation that supports analysis, optimization, and code generation, it be converted into a human-readable assembly language for debugging and visualization [2]. LLVM IR is particularly suitable for static analysis passes because it maintains complete type information, including the types of pointers.

LLVM IR uses static single assignment (SSA) form [13], which requires that every variable be defined and assigned exactly once. Using SSA form simplifies the record keeping for analyses such as optimizations because all uses of a specific definition are explicit. When

there is a branching instruction and corresponding join point in the control flow graph, the ϕ-instruction is used to select a value at the join point based on the branches taken.

Variables in LLVM IR are separate from the registers of the underlying machine. There is no limit on the number of variables that can be defined in an IR program, making it possible to add variables and expressions simply by creating new variable names.

Figure 2.1 shows a function that produces a value x that depends on the result of a function call. The corresponding LLVM IR can be found in Figure 2.2. The variable *stdin* is loaded in line 2 and stored into the variable %1. On line 3 is a **call** instruction that executes the *fgetc(...)* function with %1 as a parameter and stores the result in %2. The result is then used in the **icmp** instruction that compares %2 to 7 and stores the resulting boolean value which is of type **i1** into %3. The next instruction is a **select** instruction that will select either the 6 or 5 immediate based on %3. The **select** instruction is a special case of the ϕ instruction where there are only two predecessors that influence the value chosen. The result of **select** %4 will be returned by the **ret** instruction.

```
1 #include <stdio.h>
2 int main(){
3   int x = 5;
4   if (fgetc(stdin) > 7){
5     x++;
6   }
7   return x;
8 }
```

Figure 2.1: Simple function reading user input in C

Instrumenting LLVM IR

For our work, we instrument the LLVM IR generated when compiling a student project by adding in additional instructions. The SSA nature of LLVM IR allows us to access the

```
1 define i32 @main() {
2   %1 = load %struct._IO_FILE*, %struct._IO_FILE** @stdin, align 8
3   %2 = tail call i32 @fgetc(%struct._IO_FILE* %1)
4   %3 = icmp sgt i32 %2, 7
5   %4 = select i1 %3, i32 6, i32 5
6   ret i32 %4
7 }
```

Figure 2.2: Simple function reading user input in LLVM IR

exact parameters and return values of call instructions using the name of the associated IR variable, without being concerned about the values being changed as the program continues its execution. SSA form preserves these values, alleviating the need for making copies in the instrumentation. Using LLVM, we inserted calls into the program that record information about memory accesses and call instructions of interest. For example, consider the instrumentation shown in Figure 2.3 that calls an instrumentation function *track_fgetc* with the parameter and return value of *fgetc* on line 6. The variables that correspond to the parameter and return value can be directly passed into the function.

```
1 define i32 @main() {
2   %1 = load %struct._IO_FILE*, %struct._IO_FILE** @stdin, align 8
3   %2 = tail call i32 @fgetc(%struct._IO_FILE* %1)
4   %3 = icmp sgt i32 %2, 7
5   %4 = select i1 %3, i32 6, i32 5
6   tail call void @track_fgetc(i32 %2, %struct._IO_FILE* %1)
7   ret i32 %4
8 }
```

Figure 2.3: Simple function reading user input in LLVM IR with instrumentation

2.3 Linux Abstractions

Because the shell project is implemented for the Linux operating system, this section discusses the abstractions and interfaces provided by Linux for creating and managing user

processes as well as for facilitating communication between processes and the user, file system, and other programs.

2.3.1 Processes

In Linux, a process represents a program that is being run on the operating system. Each process has a unique identifier called a PID.

The *fork* system call creates a process. When a process calls *fork*, a new child process is created as a duplicate of the calling (parent) process. Many resources are duplicated, including the file descriptors, signal handlers, and memory.

During its lifetime, a process can be in a variety of states depending on its behavior.

Process States

- **Running** The process is either currently executing instructions or is able to execute instructions. For each CPU, the operating system's scheduler will select a process from among the processes in the **Running** state and execute it on the CPU.

- **Sleeping** The process is waiting for an event before it can execute instructions. The process may be sleeping as a result of a system call while it waits for the underlying hardware to process a request if the underlying device is much slower than the CPU.

- **Stopped** The process has been stopped and will not resume execution until it is sent a signal to do so. This process could have been stopped by another process sending a **SIGSTOP** signal or the operating system itself may have stopped the process if it detected a condition that would prevent further execution.

Processes finish execution by either calling *exit* or if the operating system or another process sends a signal that results in the termination of the process. For instance, delivery of the **SIGKILL** signal will terminate the specified process.

Signals (i.e., **SIGCHLD**) are also used to notify a process (such as the students' shell process) of when a child process has terminated or changed its process state. Since such events can occur at any time, the operating system may deliver those signals asynchronously with respect to the receiving process. A process may delay the receipt of the signal or block the receipt altogether to create critical sections of code during which pending signals are not delivered.

A user of a shell issues commands to start jobs which are composed of one or more processes. The students' shell implementation thus must maintain the current set of processes started by the shell until these processes terminate. In particular, it must inform the user when processes under its control transition to and from the Stopped state. Implementing job management in this manner requires bookkeeping on all of the created processes and managing the notifications when processes change state. Students will have to handle the **SIGCHLD** signal that notifies a process that one of its children has changed state such as when it stopped or terminated.

2.3.2 File Descriptors

A file descriptor is a handle to an input and/or output stream for a device or device-like abstraction managed by the operating system. File descriptors can refer to a variety of streams such as those representing files in the file system, a user's terminal, a network socket, or an interprocess communication channel. The descriptor is opened by a system call such as *open*, *pipe*, or *socket* and will eventually be closed by the *close* system call or when the

owning process terminates. In between the opening and closing of a file descriptor a number of operations can be performed. It is possible to *read* and *write* from the file descriptor (and thereby reading from and writing to the underlying device the descriptor represents) and to create aliases by making copies of the file descriptor. Before a file descriptor is opened and after it has been closed, no operations can be performed on it.

Since the same abstractions are provided for any file descriptor, processes do not have to be aware of the underlying specifics of the stream(s) they are using. By convention, the file descriptors with numbers 0, 1, and 2 are reserved as the standard input, standard output, and standard error output streams, respectively. The operating system allows file descriptors to be redirected, or revectored, by using the *dup2* system call. If the standard file descriptors are redirected to point to a different stream (perhaps standard output is redirected to a network socket), then the program will read from and write to the network connection represented by the socket instead. The convention is useful as it allows any program that makes use of standard I/O to work transparently even with devices that did not exist when the program was being developed.

To implement command pipelines and I/O redirection in their shell, students must understand how to redirect the standard file descriptors of the processes their shell creates.

2.3.3 Interprocess Communication via Pipes

The Linux operating system provides unidirectional pipes as a bounded buffer abstraction for interprocess communication. Pipes are primarily used by a shell to connect the output of one process to the input of another process, forming a pipeline of processes [33].

By creating all of the processes in a pipeline, the processes can self-regulate their rate of progress. As a pipe is of bounded length, a process that is producing output faster than the

subsequent process will be blocked once the amount of data written has hit the limit of the pipe. Consequently, the subsequent process can use more of the OS's resources, increasing its rate of progress until the pipe is no longer full.

A pipe is created through the *pipe* system call. This system call produces two file descriptors where one can be read from and one can be written to. When used in conjunction with the *fork* system call, the pipe file descriptors are shared between processes. The *dup2* system call is used to change the standard output of one process to point to the write end of the pipe and the standard input of another process to point to the read end of the pipe. Once the pipeline has been set up, the processes run concurrently, reading from and writing to their assigned pipes.

2.4 Features of a Shell

In this section, we will discuss the main requirements of the shell project. By understanding the requirements, we can better understand the errors made by students.

2.4.1 Jobs and Pipelines

A pipeline is a sequence of commands that are separated by the pipe (|) symbol where the output of one command is used as the input to the next. These pipelines must be created by the shell so that the data correctly flows between the individual commands and the entire pipeline is treated as a single job.

To create a pipeline, the student shell will make use of pipes (discussed in Section 2.3.3) by revectoring the standard input and output file descriptors of the processes in the pipeline.

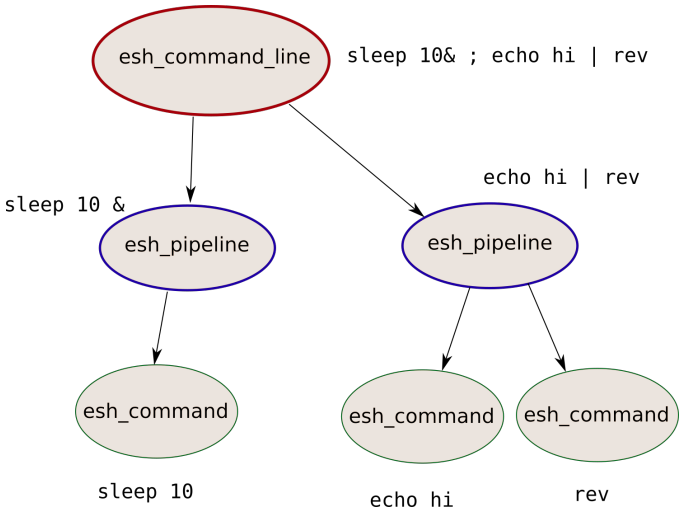

Figure 2.4: Example parse of a command line with the base shell code. This shows the command line "sleep 10 &; echo hi | rev".

2.4.2 Command Line Parsing

The shell must read the input given on the command line and parse it into an actionable form. Since the emphasis of the project is not on developing a parser, we give the students a base level of code that handles this parsing process for them and places the input into data structures. Students must learn to traverse these given data structures and use the information stored to perform the requested actions.

The data structures produced by the parser conceptually form a list of lists. For each line that is passed in, the parser produces an instance of a *esh_command_line* struct. This command line contains a list of *esh_pipeline* structs which represent the commands that are part of the same job. Each command is represented by an *esh_command* struct. As an example, the breakdown of the raw command line "sleep 10 &; echo hi | rev" is shown in Figure 2.4. This command will launch two jobs, first the command "sleep 10" in the background and then "echo hi | rev" in the foreground, shown by two separate pipelines (or jobs) with one and two *esh_command* structs as their children, respectively.

2.4.3 Job Control

The shell must provide a job control mechanism to manage all of the currently running jobs for the user. Jobs maybe considered foreground or background jobs depending on whether the user currently waits for their completion or not. Job control requires being able to monitor the state changes through which both foreground and background jobs go. Jobs that run in the background should be cleaned up in a timely manner once they have terminated, and jobs that have been stopped must be able to be resumed.

The students must implement the following built-in commands that can be executed on the jobs that the shell manages. Each of these requires managing which jobs are currently executing and sending the necessary signals to accomplish the action that the built-in specifies.

Built-in Commands:

- **kill** – terminate the specified job if stopped or running.
- **stop** – bring the specified job into the stopped state where it will remain until it is resumed.
- **bg** – run the specified job in the background.
- **fg** – run the specified job in the foreground and wait for its completion.
- **jobs** – provide a list of all the currently running or stopped jobs that are managed by the shell.

2.4.4 I/O redirection

I/O redirection allows a program to read and write directly to a file instead of to a pipe or the terminal. A user of the shell may use I/O redirection to persist the output of a program on the file system. The task for the students when implementing I/O redirection in the shell is to open the files in the correct mode (read, write, or append) and replace the standard input or output of the process appropriately through use of the *dup2* system call.

2.5 Bugs in API use

This section outlines common bugs we observed that arose from a misunderstanding or misuse of the operating system abstractions and interfaces.

2.5.1 Fork System Call

As previously discussed in Section 2.3.1, the *fork* system call allows the creation of a new process that duplicates the calling process.

A common misunderstanding students have of the duplication between the parent and child process after *fork* is that the parent's and child's memories remain aliased, so changes in one are reflected in the other. This misconception can lead to undefined behavior when the student's code assigns to variables that record child process properties, such as the PID, within the child process and intends for the parent process to be able to view the assigned value. If the parent process reads the value, it will see the previous value at the time that fork was called. This value could be undefined and could cause program crashes later on in the execution when the value is eventually used.

2.5.2 Proper Signal Handling

As discussed in Section 2.3.1, the shell must handle the **SIGCHLD** signal to be notified of the child process state changes. To implement this, students will install a signal handler in their code. A signal handler is a function that will be called whenever a signal is received. Signal handlers can be installed for a specific signal.

Signal handlers can be subject to race conditions [41] that occur when the same data is manipulated in the main program and within the signal handler. For example, in the shell project students must modify a global job data structure from within the main program to add a new job after it has been created. In the signal handler, the same data structure must be accessed and updated to record if the state of a job changes. If an element is being added to the structure when a signal is received, the structure may be in an inconsistent state. If the signal is not blocked at the time of manipulating the shared data structure, it could lead to the program crashing or data corruption.

Proper handling of asynchronous events is vital to maintaining the reliability of systems not only in the context of operating system signals but also in event-driven systems (such as mobile applications [24]) and distributed systems (such as software-defined networks [18]). By requiring proper signal handling students learn the mechanics of signals to be applied outside of the shell context.

2.5.3 File Descriptor API

We observed many mistakes students made when using the file descriptor API. One is a *file descriptor leak*, where a file descriptor that is opened is not closed once it no longer needs to be open. Such leaks take up unneeded resources which could result in running out of resources and thereby being unable to open any new file descriptors.

There are also security risks with leaked file descriptors. A child process (created by cloning with the *fork* system call) could inherit a leaked file descriptor that the child is not supposed to have access to, causing the child to be able to read/write to where they would not usually be allowed [42].

Another common mistake that occurs with file descriptors is a *double close* error. A *double close* error occurs when a file descriptor that has already been closed is closed again. A *double close* is possible because a file descriptor is represented by a small integer that could have been reused to represent a second file after being closed the first time. In this case, the second close causes adverse behavior because access to the second file would be closed unexpectedly.

A shell program makes specific usage of file descriptors to manage piping. Implementing pipes requires writing complex logic to handle file descriptors staying open as long as they are needed while ensuring that they are closed when necessary to avoid a *file descriptor leak*. The pipe file descriptors must be created in the parent process so that both the sending and the receiving process can inherit them and revector them to be their standard output and input, respectively. Since the kernel maintains a reference count for the file descriptors pointing to the pipe, any remaining file descriptors referring to the pipe's ends must be closed so that the kernel can deallocate the pipe once all of the open file descriptors pointing to the pipe's ends are closed.

2.5.4 Reaping Children

When a process is created with *fork*, the kernel allocates resources to keep track of information about that process. When a process terminates through a call to *exit* it sets a small integer known as an exit status that is communicated to the parent process. The parent pro-

cess can use the *wait* or *waitpid* calls to receive the status of a process when it is terminated to determine how it was terminated as well as the exit status of the process. Once the status of a terminated process is delivered, the kernel resources that are allocated for that process are freed. When the status of the child process is delivered and the resources associated with it are freed the child process is said to be "reaped." In the context of the shell project, it is the students' responsibility to write code that ensures that their shell reaps the child processes it starts in response to user commands.

A common problem related to reaping children occurs when the parent process fails to reap its children or delays the reaping of children unnecessarily. The result of this error is that the children and the resources associated with storing the child statuses are retained. Processes that are in the state of waiting to be reaped are referred to as zombies [23]. A significant number of zombie processes can result in performance degradation as well as an inability to create more processes.

Chapter 3

EshMD

This chapter describes the tool EshMD which is the first analysis tool developed for the shell project. This chapter is split into two sections. Section 3.1 describes the static analysis component of EshMD. Section 3.2 describes the dynamic analysis EshMD performs.

The development of EshMD was motivated by the common issues that we had seen in office hours for the shell project. Many of the issues seen required an understanding of the complex interactions of a user program and the operating system. We looked at the set of issues that could be flagged by an automatic tool and developed both static and dynamic components to aid in the students' debugging processes.

The bugs detected by EshMD fall into two categories: those related to general API misuse and those that match patterns that are specific to the application domain. The API misuse bugs represent violations of the API semantics for both the system call API as well as the API for the list data structure provided to students in the base code. The bug patterns specific to the shell project represent common issues that did not constitute a semantic violation of the API. This includes redundant variable assignments in the child process as well as failing to unblock signals after critical code sections.

3.1 Static Checkers

For the static analysis component of EshMD, we developed checkers that model the program state and look for errors. In addition to the static analysis checks that are built-in to the analyzer, we added checkers that look for errors in both the API misuse and bug pattern categories.

1. **Struct list API misuse**

   These include errors where lists were used before being initialized, attempts to remove an item not in a list, or attempts to removing items from an empty list.

2. **Redundant assignments in the child process**

   This error involves writing to a heap location without a subsequent read before calling *exec* in the child process. Since exec() reinitializes a process's memory and starts a new program, such writes are in effect dead stores whose presence is likely to indicate the fork misconception discussed in Section 2.5.1.

3. **Failing to properly unblock the SIGCHLD signal**

   This error occurs when the shell does not properly unblock the **SIGCHLD** signal when it is waiting for input. This causes the errors discussed in Section 2.5.2.

4. **File descriptor API misuse**

   This class of errors refers to a variety of errors with file descriptor management such as not releasing file descriptors, using unallocated file descriptors, and using file descriptors after they have been released. These cause the program behaviors discussed in Section 2.5.3.

By flagging these errors statically, the tool can point to code paths that suffer from these errors without requiring an execution to trigger it, allowing code paths that are not often executed (for example the signal handler paths) to be flagged and fixed by the student.

3.1.1 Using the Clang Static Analyzer

To implement these checks statically, we make use of the Clang static analyzer [10] discussed in Section 2.2.1.

The analyzer uses path-sensitive analysis to consider every path through the program individually. At each node in the abstract syntax tree, the checker modules of the analyzer are responsible for updating the state of symbolic values to model the behavior of the program and detect paths that violate each of the checker's invariants.

When the static analyzer determines that a path can lead to a state that violates the invariants imposed by the checkers, it can generate a bug report. For each erroneous path, the report shows the exact statements in the code that were traced, along with all of the assumptions that were made on the path such as which branches were taken.

We can see how the static analyzer can reason about code by looking at Figure 3.1. The code shown attempts to open a file and write data to it. If the file does not open successfully, or the data that is passed in is NULL, then the function exits early. Otherwise, the data is written to the file and the file is closed. For brevity, we ignore the checking of some system call return codes.

In this case, we are concerned with two errors, the leaking of a file descriptor and the double closing of a file descriptor.

The graph on the right of Figure 3.1 shows the control flow graph of the logData function.

```
1 // log the given data to the logfile
2 void logData(char* Data){
3     int logFD = open("logfile", O_WRONLY |
          O_APPEND);
4     if (logFD == -1) return;
5     if(!Data) return;
6     write(logFD, Data, strlen(Data));
7     close(logFD);
8     close(logFD);
9 }
```

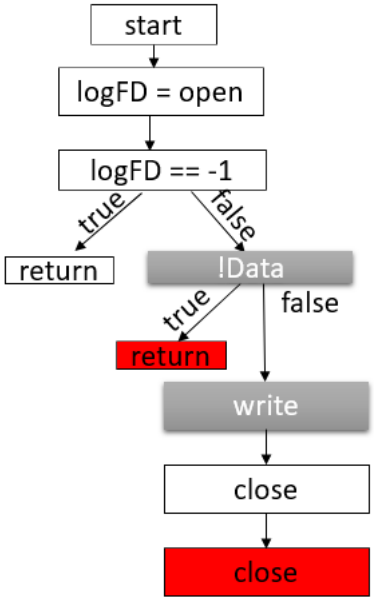

Figure 3.1: Example of tracing through code paths during static analysis.
This function suffers from a file descriptor leak and a double close error.

The job of the path sensitive checker is to determine if there is a path that ends in a leak or double close. The nodes colored gray are associated with a state where the file descriptor is considered open and the nodes in red are the detected errors.

There exists a path that will result in a return while the file descriptor is still open:

$\{(logFD! = -1) \rightarrow (!Data) \rightarrow (return)\}$.

If a null pointer is passed and the file open operation succeeds the file descriptor will be leaked.

There also exists a path that results in a double close:

$\{(logFD! = -1) \rightarrow (!Data == False) \rightarrow (write) \rightarrow (close) \rightarrow (close)\}$.

If the file is successfully opened and the data to be written is not null, the logFD file descriptor will be closed twice.

Once the buggy path is detected the path can be labeled with all of the choices that were

made for conditional expressions and return values. One labeled path will be generated for each error state that is reachable in the path-sensitive analysis. Section 3.1.3 shows the output of our tool for this example.

3.1.2 Implementation

To implement the shell-specific parts of the static analysis, we added checkers into the Clang static analysis tool. Each of these checkers requires a model to be created that captures the effects of statements on the analysis state and reports errors if a invariant is violated.

The checkers in the static analyzer work cooperatively. At each expression along the path a given checker can transition into one or more states. Then, all of the other checkers will be presented with those transitioned to states. For example, a checker that models *fork* would transition into three different states. These states represent the three possibilities of *fork* returning 0, -1, and > 0 which represent the child process, an error, and the parent process respectively. All of the other checkers will then be able to use this information on any of the paths that occur after the *fork* system call without having to model the call themselves.

Struct list API misuse

Lists must be initialized with the *list_init* function before any of the functions that use the list can be called. Also, none of the functions that remove an item from the list can be called before an item has been added.

The list API use can be modeled by looking at the list functions. Consider a list L originally in an uninitialized state. When there is a call to *list_init*(L) the list L will move into the initialized state. If any function uses the list in the uninitialized state (such as *list_size* or *list_push_back*) the list L will transition into an error state and report a use before initial-

ization error. From the initialized state, L can transition to a non-empty state if any of the functions that insert an element are called (*list_insert*, *list_push_back*, and *list_push_front*). If a list removal operation is called when the list is empty, L will transition into an error state and report an attempt to remove an items from an empty list.

Redundant assignments in the child process

A common class of mistakes involved writing to variables from within the child process and intending for that information to be communicated to the parent process. As the data segment is not shared between the parent and child process any writes in the child process will not be reflected in the parent.

We detect this error when a write to a variable without a subsequent read occurs. If the data has been written without being read, we can infer that the programmer intended to use that data in the parent process but had a misunderstanding of *fork*. We can infer this behavior because we know that a child process in the shell project should perform a minimal amount of work before it attempts to *exec*. If the value is subsequently read, we do not know if the intention was to store the value for reading in the parent, or later on in the child process, so we conservatively treat it as a valid path.

The checker models models the *fork* system call to determine the paths taken by the child process. Along the paths the child process takes, the checker identifies assignments to symbolic values and marks the value as "dirty." Uses of a symbolic value marks the value as "clean." When the path reaches an *exec* call the symbolic values are checked. If there are dirty symbolic values, the error is flagged.

Failing to properly unblock the SIGCHLD signal

The **SIGCHLD** signal allows a process to know that the state of one of its children has changed. It is necessary for the shell to handle the **SIGCHLD** signal to have an accurate record of the state of each job it manages. The shell may block the signal to make sure that the proper data structures can be updated before it is notified about a child's state change, or when it is calling *wait*. To ensure that it does not keep the children around unnecessarily long after they have terminated, the shell should unblock the signal when it performs long-running tasks (such as waiting for user input).

The checker identifies the calls made by the shell program to block and unblock the **SIGCHLD** signal. If there is a long-running operation such as an I/O operation that occurs while the signal is blocked, the error is flagged. The path that the program took to arrive at this error state will show where the signal was blocked before the program failed to unblock it.

File descriptor API misuse

The file descriptor API consists of system calls that create file descriptors (*pipe*, *open*, *creat*, *dup*, *dup2*), systems call that make use of the file descriptors (*read*, *write*, *dup2*), and the *close* system call. To model the API we initially assume all file descriptors are in a closed state except for the standard streams 0, 1, and 2. Consider a symbolic value of a file descriptor F in the untracked state. When there is a system call that creates the file descriptor, F will move into the opened state. F will remain in the opened state until a *close* system call where F will transition to the closed state. Any use or close of F must occur while it is in the opened state.

A simplified state machine that models the file descriptors that are opened is shown in Figure 3.2. The state machine will be advanced as the relevant events are encountered by

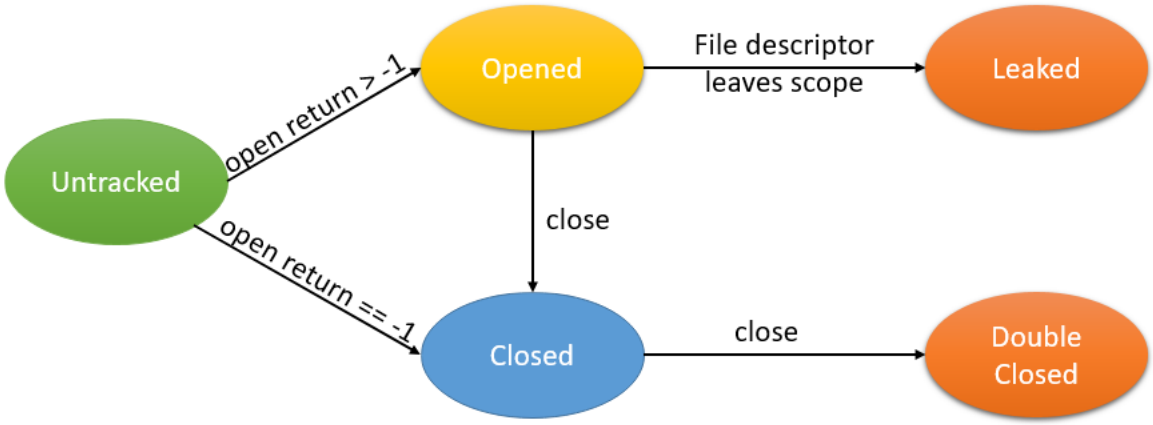

Figure 3.2: Simplified model of the states a file descriptor variable can be in. The nodes colored in orange represent error states.

the checker. The orange error states are when bugs would be reported.

If a symbolic file descriptor value is overwritten or leaves its scope, the analyzer checks for the presence of a file descriptor leak.

3.1.3 Displaying Results to Students

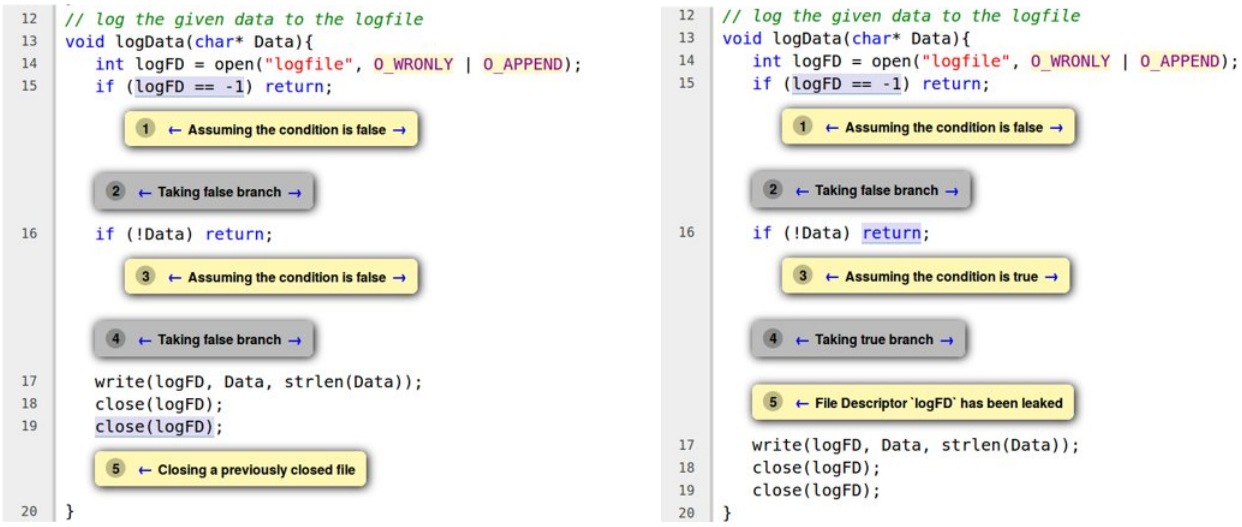

Figure 3.3: Example static analysis output shown to a student.
In this example, a file descriptor leak and a double close error are flagged.

Once a checker has reported a bug, we can trace back the path that was taken and annotate the source code to provide feedback. The analysis output that is reported to the student for the example given in Section 3.1.1 is shown in Figure 3.3. Students can use the static analysis results even if their code was not in a state that could run on all of the tests in the test driver for the shell project.

3.2 Dynamic Checkers

The dynamic analysis components in EshMD allow the tool to reason about and flag errors that occur during a specific execution of the program. They complement the static analysis by incorporating the runtime environment into the error detection, allowing the analysis to consider the actual state changes of processes under the shell's control along with which signals were received by the shell.

For this project, we execute the student submission on an existing test suite which tests all of the required functionality.

The dynamic analysis checkers that we added detect the following classes of errors:

1. Improper usage of file descriptor API

   This analysis detects the same types of errors as the static analysis, by keeping track of the current set of opened file descriptors we can flag leaks and double closes.

2. Improper process management

   This analysis monitors the state of all of the processes running on the system and can assert that the shell process waits on and sends signals only to valid child processes.

3. Redundant assignments

This analysis records the store instructions within the child process that result in redundant assignments.

4. Improper usage of list API

   This analysis detects the same types of errors as the static analysis by tracking the calls to the list API.

5. Failing to unblock SIGCHLD

   This analysis monitors the state of the SIGCHLD signal to ensure the signal is not blocked during long running operations.

To implement these checkers dynamically, we instrument the student executable to record its runtime behavior, which will be described next.

3.2.1 Instrumenting Code

The dynamic analysis relies on compile-time instrumentation, which we implemented as a LLVM compiler pass that operates on each module of code passed to LLVM. The instrumentation is added to the instructions of interest. Two common methods for instrumentation are: replacing the original instructions or placing in additional instructions. Although we could accomplish our analysis with either method, for simplicity, we add our instrumentation as an addition to the existing IR instructions at instrumentation points.

The function calls made by the program serve as instrumentation points. At each instrumentation point, we added a call instruction that triggers a call to a runtime library, passing in the necessary information to permit the detection and flagging of errors as discussed in Section 2.2.2. This runtime library, which is written in C++, is linked with the instrumented executable, avoiding function call overhead.

Since our added instrumentation passes are run before the optimization passes, we ensure that any added instructions are optimized along with the code in which they are embedded. In particular, the use of link time optimization [2] inlines most calls to our runtime library.

To better illustrate the instrumentation performed, we present an example function as it undergoes instrumentation. Figure 3.4 shows the original function in C source code. In this function, the passed in parameters are duplicated over the *stdin* and *stdout* file descriptors and closed as they are no longer required to be open.

```
1 void duplicate_pipe_fds(int pipe_fds[2]){
2     dup2(pipe_fds[0], 0);
3     dup2(pipe_fds[1], 1);
4     close(pipe_fds[0]);
5     close(pipe_fds[1]);
6 }
```

Figure 3.4: C source code setting up the pipe file descriptors for a child process

```
1  define void @duplicate_pipe_fds(i32* nocapture readonly %pipe_fds) {
2    %0 = load i32, i32* %pipe_fds, align 4
3    %call = tail call i32 @dup2(i32 %0, i32 0)
4    %arrayidx1 = getelementptr inbounds i32, i32* %pipe_fds, i64 1
5    %1 = load i32, i32* %arrayidx1, align 4
6    %call2 = tail call i32 @dup2(i32 %1, i32 1)
7    %2 = load i32, i32* %pipe_fds, align 4
8    %call4 = tail call i32 @close(i32 %2)
9    %3 = load i32, i32* %arrayidx1, align 4
10   %call6 = tail call i32 @close(i32 %3)
11   ret void
12 }
```

Figure 3.5: LLVM IR setting up the pipe file descriptors for a child process

The corresponding LLVM IR that is generated for this function is shown in Figure 3.5. The call instructions that correspond to the function calls seen in the C code of Figure 3.4 occur at lines 3, 6, 8, and 10. The other instructions deal with fetching the correct element of the parameter array to pass as parameters to the call instructions.

```
1  define void @duplicate_pipe_fds(i32* nocapture readonly %pipe_fds){
2    %0 = load i32, i32* %pipe_fds, align 4
3    %call = tail call i32 @dup2(i32 %0, i32 0)
4    tail call void @dup2Called(i32 %0, i32 0)
5    %arrayidx1 = getelementptr inbounds i32, i32* %pipe_fds, i64 1
6    %1 = load i32, i32* %arrayidx1, align 4
7    %call2 = tail call i32 @dup2(i32 %1, i32 1)
8    tail call void @dup2Called(i32 %1, i32 1)
9    %2 = load i32, i32* %pipe_fds, align 4
10   tail call void @closeCalled(i32 %2)
11   %call4 = tail call i32 @close(i32 %2)
12   %3 = load i32, i32* %arrayidx1, align 4
13   tail call void @closeCalled(i32 %3)
14   %call6 = tail call i32 @close(i32 %3)
15   ret void
16 }
```

Figure 3.6: Instrumented LLVM IR setting up the pipe file descriptors for a child process ed analyses

In Figure 3.6 the same LLVM IR code is shown after the instrumentation passes are run. At each of the highlighted lines, is added in a call to our runtime library. The instrumentation pass directly passes the parameters and return values of the functions of interest to the runtime library.

3.2.2 Implementation

This section details the specific implementation of the checkers that are responsible for flagging errors dynamically, describing the specific behaviors that are modeled and the errors that are flagged.

Modeling File Descriptors

To trace file descriptor API usage, we instrument every function call that deals with file descriptors (*dup2*, *pipe*, *close*, and *open*) as well as the functions that create and destroy standard I/O `FILE` objects (*fopen* and *fclose*). We included *fclose* and *fopen* when we saw that students sometimes used `FILE` objects in conjunction with the *fileno* function to implement I/O redirection, presumably because they were more familiar with these calls than the *open* system call.

During execution, we maintain a table of the file descriptors that have been opened or closed. Whenever a file descriptor is used, we check against this table to determine if this file descriptor operation is operating on valid file descriptors. We also keep a stack trace of when a file descriptor was opened or closed, which is reported if error conditions are detected later.

Another error that can be detected is leaking opened file descriptors to the child process. One issue this causes is that the child process will continue to use additional resources unknowingly and wastefully. This problem could make itself even worse if the child process called *fork* to spawn children of its own because those grandchildren would inherit their own copies of the leaked descriptors.

In the shell context, leaking file descriptors can also cause correctness errors. Programs such as **cat** or **wc** are designed to read from their *stdin* stream until there is no more data to be read, i.e. an EOF condition is encountered. If those programs are used at the receiving end of a pipe, the EOF condition is signaled if and only if all file descriptors that refer to the other end of the pipe have been closed. If a file descriptor referring to the write end of a pipe is leaked and thus not closed, this condition will never occur, leading the program to hang or "getting stuck." This condition can occur both in a child process that is spawned

by the shell as well as within the shell itself.

In the child process, we can detect this bug by instrumenting the *exec* family of system calls. When *exec* is called, we can check all of the currently opened file descriptors. Any file descriptors that are still open, except for the standard file descriptors which we expect to be open, will be appropriately flagged and reported to the student. In the parent process, we instead instrument the *wait* and *waitpid* calls and flag opened pipe file descriptors when a blocking *wait* is called. We ignore non-blocking waits (i.e., *wait* called with the `W_NOHANG` flag). In a correct implementation, such calls are present in the **SIGCHLD** signal handler. Ignoring such calls prevents false positives when a **SIGCHLD** signal is delivered and processed for a background job after the parent process has already created the pipe, but before it called *fork* to start the child process.

Tracing process-related events

The dynamic analysis monitors the PIDs and associated process groups of the processes and process groups created by the shell. This ensures that signals sent by a *kill* system call are not being sent to a process that either no longer exists or to a process that has never been created. New PIDs are learned using the PID values returned from the *fork* call in the parent process whereas a successful return of a *wait* call signals the retirement of a PID whose process was terminated.

The shell must set up process groups as part of the job control mechanism to allow for the proper sending of signals to all processes comprising a pipeline. For instance, a built-in command such as *stop* sends the **SIGSTOP** signal to the entire process group, which will be delivered to all individual processes in the group. We can monitor the correct handling of process groups by instrumenting the *setpgrp*, *setpgid*, and *kill* calls issued by the shell. A

process group can only be created from either of the *setpgid* or *setpgrp* calls, and is destroyed once all of the processes inside of the group have terminated. Our analyzer flags attempts to send signals to a process group whose identifier that has not been recorded (because the shell failed to issue the necessary calls to create it). Such attempts not only indicate a bug in the student's shell, but could lead to the unintended delivery of signals to process groups using the identifier in question.

Process groups are also used to manage access to the terminal. Our analyzer detects attempts to give possession of the terminal to a process group that does not exist.

Identifying redundant assignments

This dynamic analysis checker monitors every load and store instruction that is made by the child process before *exec*. On store instructions, the address stored is marked as modified, and on a load to the same address, it is marked as clean. When the child process calls *exec*, we can check what addresses, if any, have been marked as modified and report an error.

Modeling lists

This dynamic analysis checker monitors the struct list by instrumenting every list API call. This analysis uses the same model for the lists as the static analysis (discussed in Section 3.1.2) to flag the use of an uninitialized list and the removal from an empty list errors.

Tracing SIGCHLD blocking

This dynamic analysis checker traces the calls to *sigprocmask* to determine when the SIGCHLD signal is blocked. To flag errors when SIGCHLD is blocked on long running processes, the

checker traces the blocking *wait* call used to wait for child processes and the *readline* call used to read user input from the terminal.

3.2.3 Displaying Results to Students

Figure 3.7 shows how our tool reports error locations using their associated stack traces. Students are already familiar with stack traces from debugging exceptions in languages such as Java or Python, or from using GDB [19] in their prior coursework.

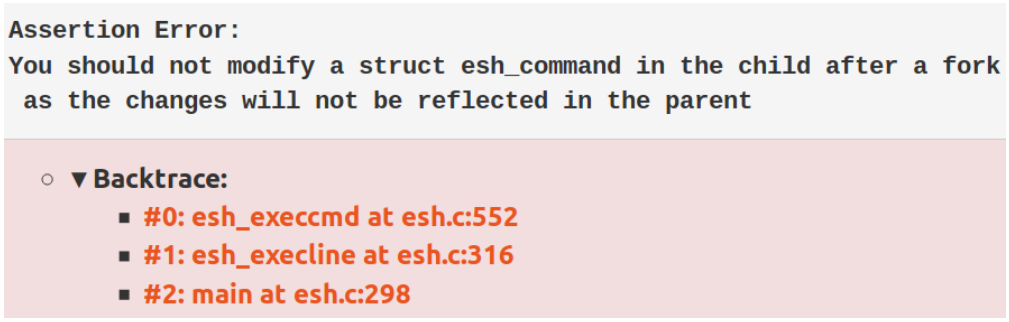

Figure 3.7: Stack trace shown to students when the dynamic analysis detects an error

Figure 3.7 shows an example of the stack trace shown. Each source location in the stack trace links directly to a highlighted rendering of the student code (seen in Figure 3.8), pointing out the call within the context of the surrounding code.

```
547             pid_t pid;
548             pid = fork();
549             if(pid == 0)
550             {
551                 jid= getpid();
552                 command -> pid = getpid();
553
```

Figure 3.8: Student source code highlighted to indicate the line of interest.

Running the instrumented executable

A supplemental benefit to performing instrumentation at compile time is that the instrumented executable, which contains all analysis code, can subsequently be provided to the students.

Providing the executable allows students to run the analyses on their own test cases or the standard test cases to see the situation where the analyzer reports errors. They can also diagnose less frequently occurring bugs by running the instrumented executable multiple times. We compile the instrumented executable with debugging information to allow the student to use GDB. This can be useful for pre- or postmortem debugging when an error is detected.

3.3 Applying EshMD to Other Projects

While EshMD was developed for the shell project in its current incarnation, its concepts and implementation could be generalized and applied to other projects, which would make the tool applicable to future versions of the shell project and potentially other assignments.

First, some of the checkers in EshMD are already broadly applicable such as those that detect misuse in the file descriptor API, which is not specific to the shell project. Any project that requires the use of file descriptors would benefit with no effort from these checks.

Second, to use EshMD on a new project or application, expected bug patterns or undesired behaviors and their causes must be modeled and embedded into EshMD. This requirement makes EshMD most appropriate to school assignments where the common errors can be observed in the past offerings of that assignment.

Chapter 4

ShellTrace

ShellTrace was developed after applying EshMD to past student submissions, which showed that certain bug patterns were not detected by EshMD. These patterns included missing API calls, unnecessary API calls, and making use of the API in ways that violate the project specification, but do not violate the models EshMD assumes.

Examples of each of these types of errors included:

- Never calling *pipe* when processing a pipeline command.

  The *pipe* call is required to facilitate interprocess communication between the processes of a pipe. When the *pipe* call is missing, the processes will not pass their output through pipes and will instead read from and write to the terminal directly.

- Creating additional, unused pipes.

  Although these errors could manifest as leaked file descriptors, we observed that students sometimes added calls to *close*. As such, the file descriptor API semantics is not violated, but the pipe itself is redundant, wastes systems resources, and indicates confusion on the part of the student.

- Placing all spawned processes into the same process group.

  Proper job management requires that each process be placed into its job's process group. Some students, however, placed all processes from all jobs into a single process

group. This behavior violates the project specification because stopping a single job via a signal sent to its process group should not stop all jobs the shell controls.

The core idea of ShellTrace is to automatically derive a specification from a trace of relevant call events that is generated by a reference solution, then to determine if the student solution meets that specification. Since there are many orderings of calls that produce the correct result, a simple matching of traces between the reference and student solution is insufficient. Instead, ShellTrace derives all possible orderings of calls that produce the same result as the reference solution. These orderings are represented in the form of a graph, which can be thought of as forming a program specification.

The goals of ShellTrace are the following:

1. Automatically derive a program specification from executing a reference solution.
2. Point out where the student solution deviates from the project specification in significant ways that are likely to affect correctness and/or efficiency.
3. Visualize the behavior of their shell along with where it significantly differs from the reference solution (without displaying the exact sequence of calls made by the reference solution).

4.1 Design

When determining whether two calls that occur in the reference solution's trace can be reordered or not, ShellTrace considers the resources that are created, accessed, or destroyed by the API to which the call belongs. Examples of such resources include file descriptors and process identifiers. Thus, the specification derived from the reference solution embodies

the required interactions with operating system resources.

ShellTrace considers a trace of function calls made by the program and models them as resource actions. By modeling the dependency relationships between actions that refer to the same resource, ShellTrace can determine if calls can be reordered while producing identical functionality. The reference and the student implementations can be compared to see if the student implementation satisfies the same dependency relationships as the reference implementation, making their functionality identical.

Table 4.1: Types of Resource Actions in the ShellTrace Model

Resource Action	Meaning
Creation(T,I)	A resource with type T and identifier I has been created.
Use(T,I,[un]ordered)	A resource with type T and identifier I has been used.
Release(T,I)	A resource with type T and identifier I has been released.

Table 4.1 shows the types of resource actions considered in the ShellTrace model. A type and identifier tuple parameterize each action. Each resource has a separate namespace for its identifiers, allowing us to distinguish between different resources that may share the same identifier, such as file descriptors and process identifiers.

We discuss the resource actions in more detail below.

Detailed Breakdown of Resource Actions:

Creation(T,I) The *Creation* action corresponds to the creation of a resource, such as a call to *open*. Once the program creates a resource it can be used until it is released. Each call results in a new resource that has an identifier that is separate from the previous resources.

Use(T,I,[un]ordered) The *Use* action has an additional parameter *[un]ordered* which can

take the value of *ordered* or *unordered*. For both values of *[un]ordered*, the resource must be created before the *Use* can occur. A *Use* action is dependent on the previous actions that make a change to the underlying resource which includes the *Creation* and ordered *Use* actions.

Ordered *Use* An ordered *Use* action represents some modification of the underlying resource, such as advancing a file pointer in a *read* operation. This modification means that any subsequent action will be dependent on this *Use*.

Unordered *Use* The unordered *Use* represents an action that uses the current state of the resource without modification. An unordered *Use* action will not be dependent on other unordered *Use* actions. The unordered use will still be dependent on and cause the same dependent relationships as the ordered *Use* for all other actions.

Release(T,I) A *Release* action represents the consumption of a previously created resource. Subsequent *Creation* actions may then reuse the same type and identifier combination. Some resources may not have explicit *Release* actions and instead will be implicitly consumed when the program terminates.

A dependency graph is created from a sequence of these resource actions. This dependency graph will be a directed acyclic graph (DAG). The nodes and edges in this DAG are defined as follows:

Nodes The nodes in the graph represent the resource actions that the program performs during an individual execution. All monitored actions generate a node in the graph. The nodes contain the type and identifier parameters based on the call that corresponds to the resource action(s).

Edges The directed edges in the graph occur from a parent or source node to a child or destination node. An edge is created when the child node is directly dependent on the parent due to a semantic dependency. There exists an edge from node A to node B if the action B represents cannot occur before A while still producing the same functionality.

The graph must be acyclic because there cannot be a circular dependency since that would mean the actions could not have occurred in the original execution. As there are no dependencies between different type-identifier pairs, the graph may be disconnected. For ease of visualization and comparison, we connect the graph by introducing a special start node that does not correspond to a resource action but serves as the immediate parent to all nodes without parents.

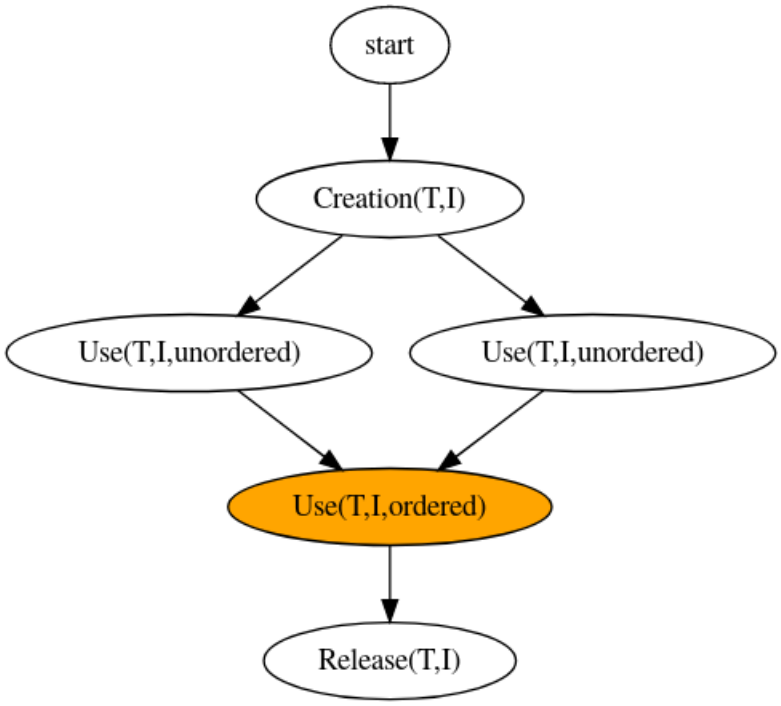

Figure 4.1: Resource action dependency graph demonstrating ordered Use actions

Figure 4.1 shows an example dependency graph drawn from the following sequence of actions

with type T and identifier I:

1. Creation(T,I)
2. Use(T,I,unordered)
3. Use(T,I,unordered)
4. Use(T,I,**ordered**)
5. Release(T,I)

The unordered *Use* actions are dependent on the *Creation* action and may appear in any order relative to each other in an execution. The ordered *Use* (colored in orange) must occur after the preceding unordered *Use* actions. Finally the *Release* action must occur last.

To model the shell project, we provide the mapping between the system calls and their respective resource actions. By modeling the relevant system calls, we can capture a complete picture of all of the dependent resource actions required for the shell project.

Table 4.2: Calls and events paired with resource actions.

Function call or Event	Resource Action(s)
open function call	Creation(file descriptor, return value)
pipe function call	Creation(file descriptor, parameters)
read function call	Use(file descriptor, first parameter, ordered)
write function call	Use(file descriptor, first parameter, ordered)
dup2 function call	Use(file descriptor, first parameter, unordered), Creation(file descriptor, second parameter)
fork function call	Use(file descriptor, all values, ordered), Creation(process, return value)
waitpid function call	Release(process, return value)
wait function call	Release(process, return value)
getpid function call	Creation(process, return value)
killed by signal	Release(all types, all values)

Table 4.2 shows a list of the events and system calls paired with their appropriate resource actions. A system call may correspond to multiple resource actions based on its semantics. The *fork* system call is special because it causes a duplication of resources such as file descriptors into both the parent and child processes. This effectively places a barrier into the trace where calls on either side of the *fork* call cannot be reordered with respect to the *fork* call.

Figure 4.2 shows an example of a dependency graph generated from the following trace of system calls which map to resource actions:

1. 3 = open(...) → *Creation* (file descriptor, 3)
2. dup2(3, 5) → *Use* (file descriptor, 3, unordered), *Creation* (file descriptor, 5)
3. dup2(3, 6) → *Use* (file descriptor, 3, unordered), *Creation* (file descriptor, 6)
4. read(6, ...) → *Use* (file descriptor, 6, ordered)
5. close(3) → *Release* (file descriptor, 3)

The *dup2* calls are dependent on the *open* call as *open* creates the file descriptor 3. The *close* call must happen after the *dup2* calls, or the *dup2* would return an error and not create a new file descriptor. The *read* call can happen at any point after *dup2*(3, 6), this could be before or after both *dup2*(3, 5) and *close*(3).

4.1.1 Reporting Bugs in an Implementation

For ShellTrace, one goal was to model the student program's execution so that the student could identify calls where their program's execution differed from the project specification.

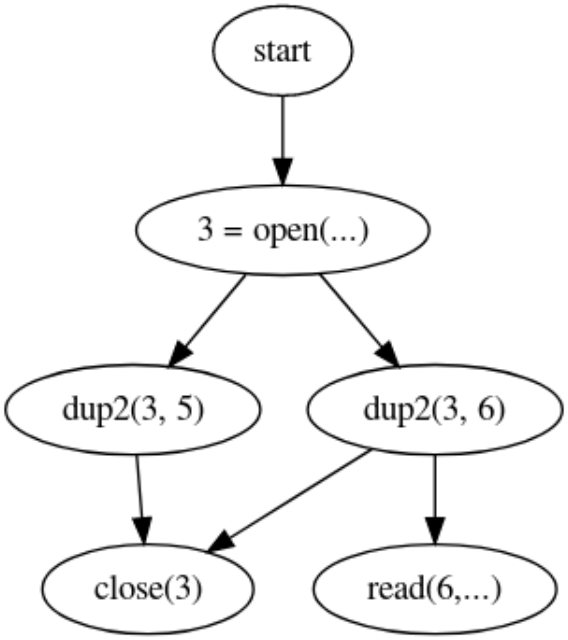

Figure 4.2: Dependency graph for example trace of system calls

To facilitate this, ShellTrace creates the graphs of dependent resource actions from the model described in Section 4.1 for both the student submission and the reference solution. We compare the generated graphs of the two program executions and determine if they possess different dependent relationships. These differences typically will be the presence of a node or collection of nodes in one of the graphs but not the other. Because ShellTrace models all required functions, a difference in the dependency relationships is evidence of a difference in their observable functionality.

When the student solution has an additional node in the graph, the call that generated that resource action is unnecessary. When the student solution is missing a node the solution is not correctly using operating system resources to implement the shell functionality.

We alert students to the differences between their implementation and the reference solution by highlighting unnecessary nodes or placing the next necessary (but missing) node at the appropriate point in their graph. The student can see why the flagged node is necessary or

unnecessary by looking at the surrounding dependencies.

4.2 Implementation

This section defines the implementation details of ShellTrace, with a particular focus on how its techniques could be reused if the tool were applied in other projects. We discuss the implementation of our graph comparison algorithm in Section 4.2.1 and the specifics of configuring the mapping of specific program events to resource actions for the ShellTrace model in Section 4.2.2.

4.2.1 Comparing ShellTrace Graphs

To compare the resource dependency graphs generated by the reference solution and the student's solution, we identify the nodes that are present in one graph but not the other. Additionally, we identify the minimum difference between the two graphs which we define as the minimum number of node additions and deletions that must be made to one of the graphs to make the two graphs isomorphic.

Although determining whether two graphs are isomorphic is not known to be solvable in polynomial time for general graphs [4], we can take advantage of the nature of our specific problem to make the comparison feasible.

As described previously, the graph created by ShellTrace is directed, acyclic, and connected. Also, a ShellTrace graph has a single starting node that must occur before any actions. Using these properties, our algorithm begins at the starting node of each graph and finds the matching of child nodes that results in the minimum number of required changes between the graphs.

Suppose we want to find the minimum difference between two ShellTrace graphs. The minimum difference between the graphs is equal to the minimum difference between the start nodes in the graphs. The minimum difference between two nodes is defined as the sum of the difference between the nodes and the minimum difference between all possible child node assignments. A child node assignment is a matching between the child nodes of a node in one graph to a node in the other graph and empty nodes such that each child node is matched with one other child node or an empty node. The difference between two nodes of the same resource action is zero, the difference between a node and an empty node is one, and the difference between two nodes of different resource actions is two.

For two graphs G and G' of size n and m, $n \geq m$, the algorithm will consider the node pairs between all nodes that are at the same distance from their respective start nodes. $D(i, g)$ represents the number of nodes at distance i from the start node in graph g.

The total number of pairings p to consider is $p = \sum_{i=0}^{n} D(i, G) * D(i, G')$.

For each of the p pairings, computing the minimum difference between the two nodes requires computing the assignments of all of the child nodes. Let $C(x)$ represent the number of children of node x. The nodes y and z in the pairing, $C(y) \geq C(z)$, will have no more than $a = C(y)! * 2^{C(y)}$ child assignments. Each of the child node matchings in one of the $C(y)!$ assignments can either be present (the child nodes are matched with each other) or not present (both child nodes in the matching are matched with empty nodes), resulting in $2^{C(y)}$ choices for a given assignment.

The total number of pairs is upper-bounded by the total number of nodes in the graph $n*m$. The number of child assignments is bounded by the maximum number of child nodes A. This makes the overall time complexity $O(n * m * 2^A * A!)$.

In practice, on the shell project, the maximum number of children for any node was three.

The nature of many of the modeled APIs produces few children for an individual node because an ordered *Use* and a *Release* will restrict themselves to be the only child for that given resource type. This results in no more than $A = 3! * 2^3 = 48$ possible assignments for each pair of nodes. By considering the amount of child assignments between nodes constant, the complexity becomes $O(n * m)$.

4.2.2 Configuring ShellTrace

ShellTrace is implemented similarly to the dynamic checkers in the EshMD tool, which are described in Section 3.2.2. ShellTrace consists of a LLVM pass that instruments the relevant points in the submission with calls to a dynamic library which provides functions to handle each resource action. As a given call such as *dup2* could correspond to multiple different actions (an unordered *Use* for the first parameter and a *Creation* for the second parameter) the library must be able to merge these actions into the same node. At instrumentation time, the call that generated the instrumentation is recorded so that the runtime library can merge resource actions generated by the same call into the same node.

Unlike EshMD, where the instrumentation is tailored directly to the bugs detected, ShellTrace has configurable instrumentation points. Our initial prototype implementation considers the resource actions listed in Table 4.2. These are provided as a configuration file to ShellTrace; other call to action mappings could be added without needing to change the dynamic runtime library. The configuration file is read by a LLVM pass before it begins to instrument the submission.

An example configuration file that demonstrates relevant excerpts is shown in Figure 4.3. This configuration file defines the signals of interest to be **SIGINT** and **SIGTERM** and provides objects describing the mappings for 2 file descriptor related system calls, along with

descriptions of the semantics of their positional arguments and return values. Each object in the `functions` field has an optional `args` (arguments) property and a `returns` property which correspond to the resource actions that the arguments and return value generate, respectively. Each of these objects has an `action` property that contains the name of the runtime library function that will be called by the instrumentation logic. The *Use* action requires an additional `ordered` field that flags it as being ordered or unordered.

Since ShellTrace reads this configuration file at instrumentation time, the code must be re-instrumented after a configuration change if a different set of actions is to be modeled.

```
{
   "functions":[
      {
         "name":"open",
         "returns":{
            "type":"fd",
            "action":"creation"
         }
      },
      {
         "name":"dup2",
         "args":[
            {
               "type":"fd",
               "action":"use",
               "ordered":false
            },
            {
               "type":"fd",
               "action":"creation"
            }
         ]
      }
  ],
  "signals":[
    "SIGINT",
    "SIGTERM"
  ]
}
```

Figure 4.3: Example ShellTrace Configuration

Chapter 5

Evaluation

We deployed the analysis tools EshMD and ShellTrace in the undergraduate computer systems course at Virginia Tech (CS 3214) for the Spring 2018 semester. The submissions made to the analysis tools were all recorded anonymously. A post-survey was released to the students after project completion. The data was collected with approval from the Virginia Tech Institutional Review Board under research protocol IRB#17-032.

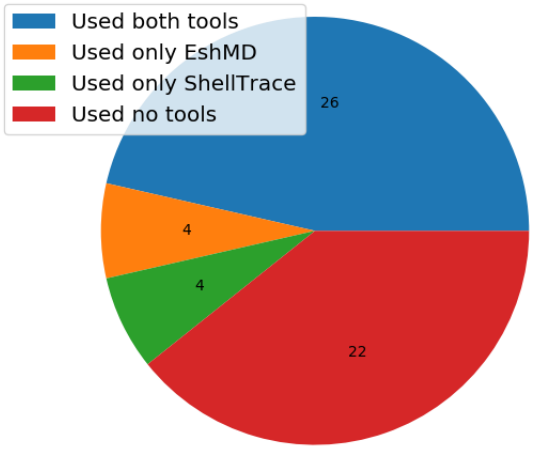

Figure 5.1: Student survey responses on analysis tool use

For this study, we had 85 students submit their code to the analysis system. For the survey, we had 56 responses where 34 students indicated that they had made use of the analysis tools. There were 182 students that submitted the project for grading, giving a 31% response rate. Figure 5.1 shows the responses to the survey question asking which tools the students

used. Since each a submission was run against both of the analysis tools and the results of both tools were indicated with links on the students' personal submission page, it was easy for students use both analysis tools if they desired. From the responses, we can see that a plurality of respondents actively used both EshMD and ShellTrace rather than a single tool.

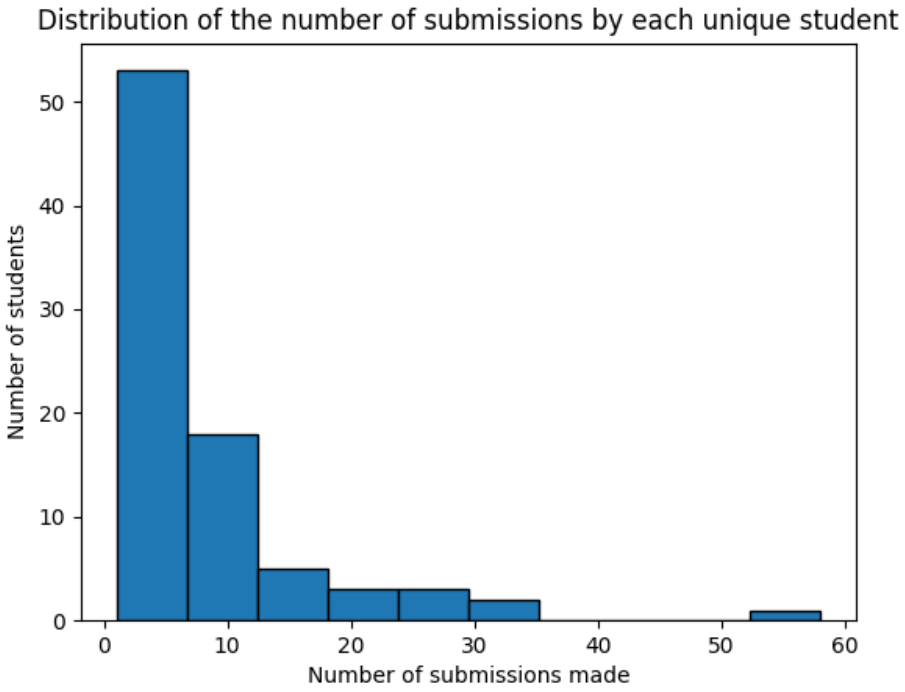

Figure 5.2: Number of submissions by each student to the analysis tools

Of the students that submitted to the tools, we had an average of 7 submissions per student with a median of 4 submissions and a maximum of 58 submissions. The distribution of the frequency with which students submitted to the tool is shown in Figure 5.2. While the majority of the students made very few submissions, there were a few students who made extensive use of the tools by submitting many iterations of their solution.

In the survey, students were asked to provide their reasoning for why they did not make use of the tool. The following are representative quotations of the feedback received:

- "My partner and I made the mistake of not being thorough in our reading of the project spec. We were simply not aware of these programs' existence."
- "I didn't get around to it. My code worked and it passed the suite of tests I was given."
- "I felt confident debugging with other tools."

In the future deployments of the tools, we can reiterate their existence periodically in both class lectures as well as the project specification as students work on the project. Further improvements to the tool explanations and usability could encourage more students to understand and make use of them.

In Section 5.1 we evaluate EshMD and in Section 5.2 we evaluate ShellTrace.

5.1 EshMD

5.1.1 Errors Detected by EshMD

As this project was originally inspired by anecdotal evidence of common bugs observed when helping students with the project, we assessed how often each type of bug appeared in actual student code. We assessed this in two ways: by looking at all of the errors that the dynamic analysis of EshMD detected across student submissions and by surveying the students to determine what errors they thought were flagged by the tools.

The error types we identified for EshMD are below.

EshMD detected error types:

- **file descriptor** These errors deal with the improper management of file descriptors including the leaking of file descriptors and the use of closed file descriptors. These errors involve both file descriptors opened for I/O redirection and file descriptors opened by pipes. The most common errors within this class were leaving file descriptors of pipes open within the child process and attempting to use a pipe that had already been closed.

- **pgid** These errors deal with the improper management of process groups. When the shell starts a pipeline of commands, all of the commands must be placed into the same process group. The most common error concerning process groups was not setting the process group before attempting to perform an action on that group such as giving that process group the terminal or sending a signal to that process group.

- **child assignment** These errors deal with redundant assignments to variables in the child process before *exec*, as discussed in Section 2.5.1.

- **list** These errors involve any misuse of the provided list API functions including not initializing lists as well as incorrect iteration and modification of lists. While this error was anecdotally seen a significant number of times in both office hours and questions on the class forum board, there were very few implementations submitted that suffered from this issue. We believe this is due to the assertion checks within the list code which would trigger an error for the students before they submit their code for analysis.

- **signal blocking** These errors are due to students failing to unblock the **SIGCHLD** signal once the critical section is completed which could result in "zombie" processes, as discussed in Section 2.5.4.

During dynamic analysis, the student submission was run against the provided test suite. The test suite is divided into two categories: tests that focus on proper job control and tests

that require piping and I/O redirection to be implemented. Each EshMD dynamic analysis checker focused on one error type and each submission was run individually to determine if any of the test cases run suffered from that type of error. An individual submission could contain errors from multiple error types. Multiple errors of the same type that were detected in the same test suite category are counted as a singular occurrence.

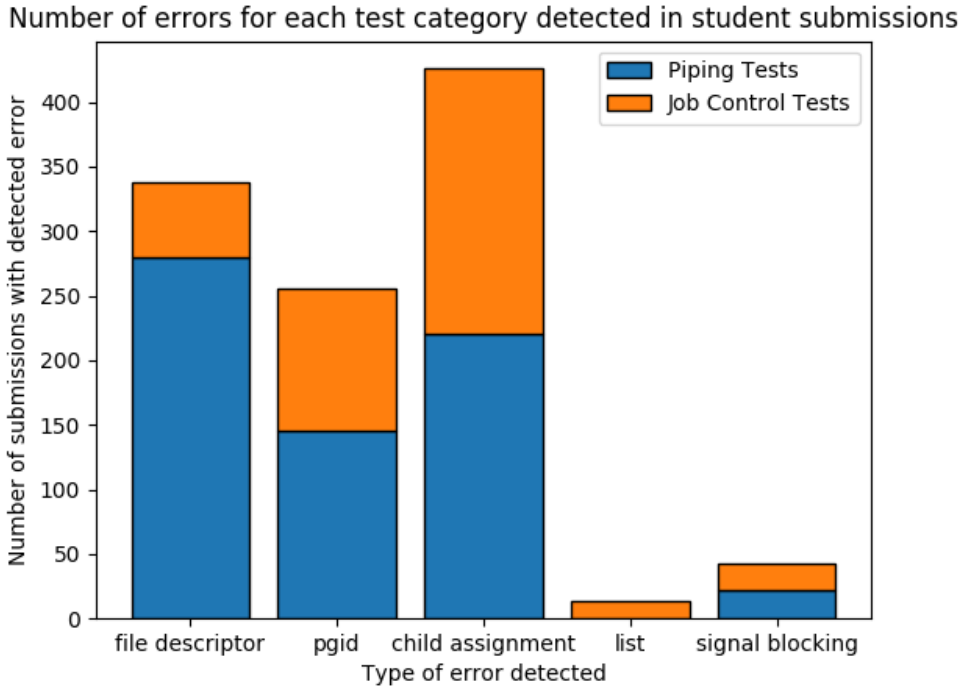

Figure 5.3: Error breakdown by test suite category in EshMD

Figure 5.3 shows the breakdown of the error types by the test suite category. The **file descriptor** errors occur mainly in the piping tests as these tests require logic that deals with many file descriptors. The **pgid** errors occurred equally across test categories as both require that jobs are properly placed into their respective process groups. The **child assignment** errors appeared equally across the test categories because the errors flagged would be on the common case path of executing a job. The **list** errors occurred infrequently and only in the job control tests where students would attempt to add to lists that were not initialized. The

signal blocking errors also occurred infrequently as students were shown how to correctly block SIGCHLD in a help session for the project.

As part of the post-survey, students were asked to identify bugs that the EshMD tool flagged in their submissions. This question was used to evaluate the student perception of the tool and how that matched up with the actual bugs found in student submissions. Students were asked to identify errors that EshMD found in their submissions and the errors reported by the students were mapped into the appropriate EshMD error types. Figure 5.4 shows the student survey responses.

From Figure 5.4 we can see that the errors students reported most were those that fell into the **file descriptor** type which reinforces our anecdotal evidence of file descriptor API misuse contributing to student bugs. The **list** error type was also encountered by a large set of students but was not repeated in subsequent submissions, so the students must have fixed the error after it had been encountered the first time. The **pgid** and **child assignment** types had almost no student reports. This result may be because if errors of this type were present in a submission, they would be flagged in every test, making it likely that students fixed the bug early and then did not remember it.

5.1.2 Effectiveness of EshMD

Students were also asked to evaluate the usefulness of EshMD. The survey results for the evaluation of EshMD is shown in Figure 5.5. These results indicate that not only did EshMD find bugs in the student submissions but that our goal of helping students discover and fix these bugs was realized.

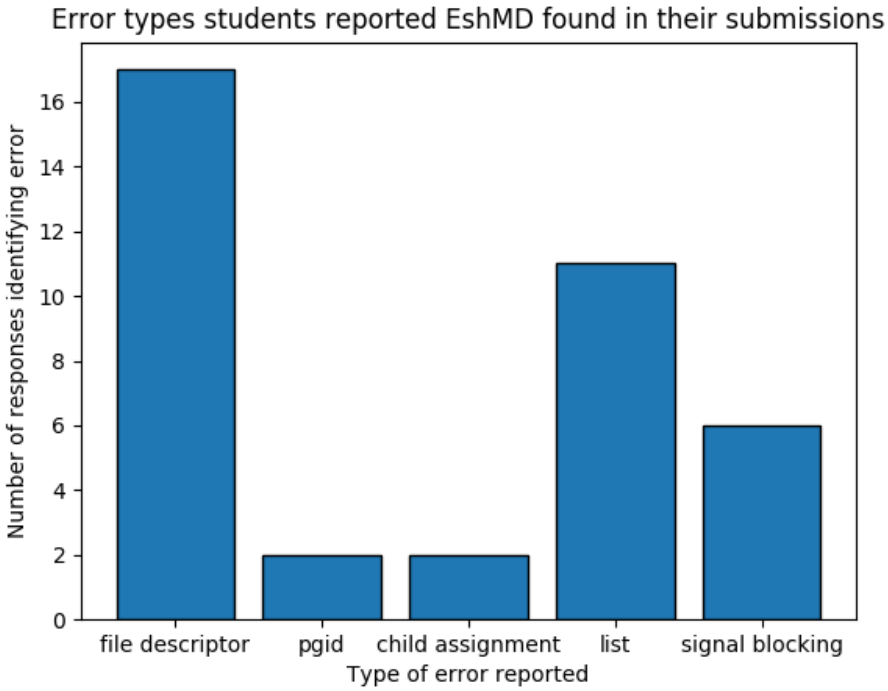

Figure 5.4: Error classes students identified as flagged by EshMD

5.2 ShellTrace

5.2.1 Errors Detected by ShellTrace

Like EshMD, ShellTrace was designed to help students diagnose errors in their implementations. The error types for ShellTrace reflect the errors that were displayed to the students as feedback. If the student solution created a different number of processes than the reference solution the tool reported the difference to the student. If the number of processes matched, ShellTrace compared the generated graphs and reported the unnecessary or missing system calls.

The error types for ShellTrace are as follows:

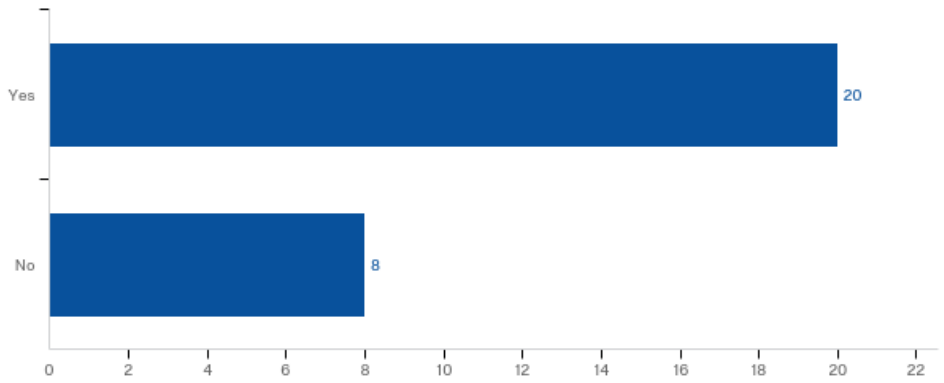

Figure 5.5: Survey results for "Did you find EshMD helpful in detecting errors?"

ShellTrace error types:

- **Fork Mismatch** This error type includes all errors where the student solution did not call *fork* the same number of times as the reference solution.
- **Missing Call** This error type includes all errors where the student solution did not perform all calls of the reference solution required for correct execution.
- **Unnecessary Call** This error type includes all errors that involve a student solution performing unnecessary or erroneous calls that are not present in the reference solution.

Figure 5.6 shows how many bugs of each type were detected in student submissions. The submissions were run on all of the shell test cases. The reported error is the first error for a given test that is encountered. The first error encountered was the error that was given as feedback to the students.

Figure 5.6 shows that the largest error type for ShellTrace was due to the mismatch in the number of fork system calls between the reference solution and the submission. Nearly all of these ($> 90\%$) were due to the student submission calling *fork* less than the reference solution. This result is due to the submission crashing due to a segmentation fault or exiting

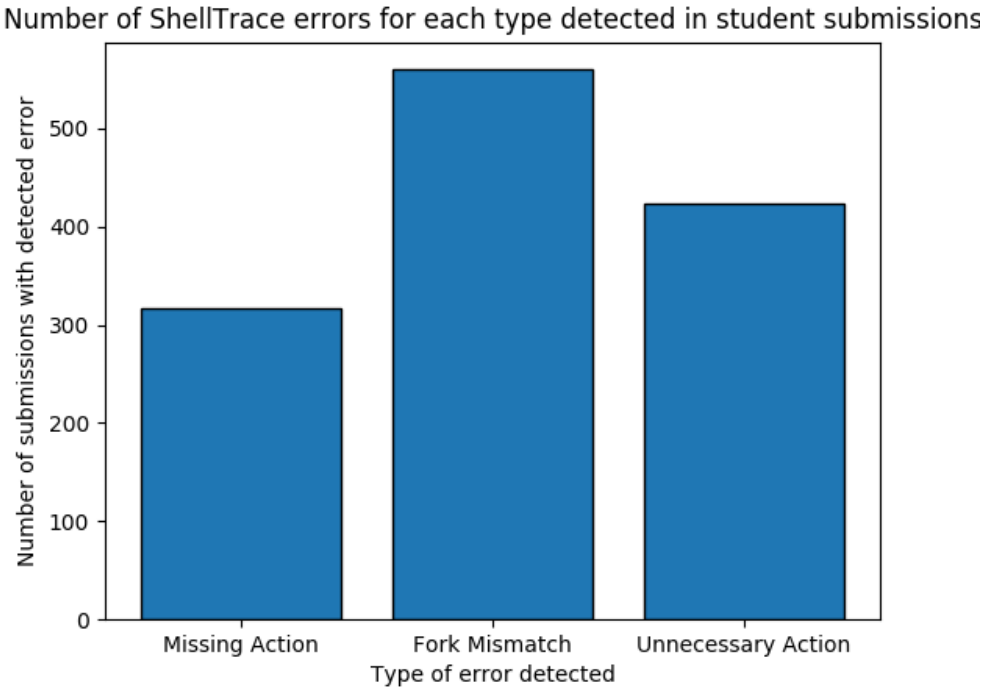

Figure 5.6: Errors detected by error type in ShellTrace

due to an assertion failure. This result was unexpected as we had made design choices for the tool that assumed the student submission would run to completion. Traditional debugging tools such as GDB [19] would serve a better purpose in debugging crashes due to memory access or assertion errors than ShellTrace.

The next largest type of errors was Unnecessary Call. Figure 5.7 shows a further breakdown of the unnecessary calls in the student solutions. We see there are some file descriptor related calls such as *pipe*, *open*, *close*, and *dup2* which occur when the student solution opens up more file descriptors than needed or calls *close* more than necessary. Based on our experience, we expected file descriptor related calls to be the majority of errors.

We did not expect a large number of unnecessary calls to *getpid* from this error type. When we sampled the submissions, we found that these calls corresponded to a subset of the

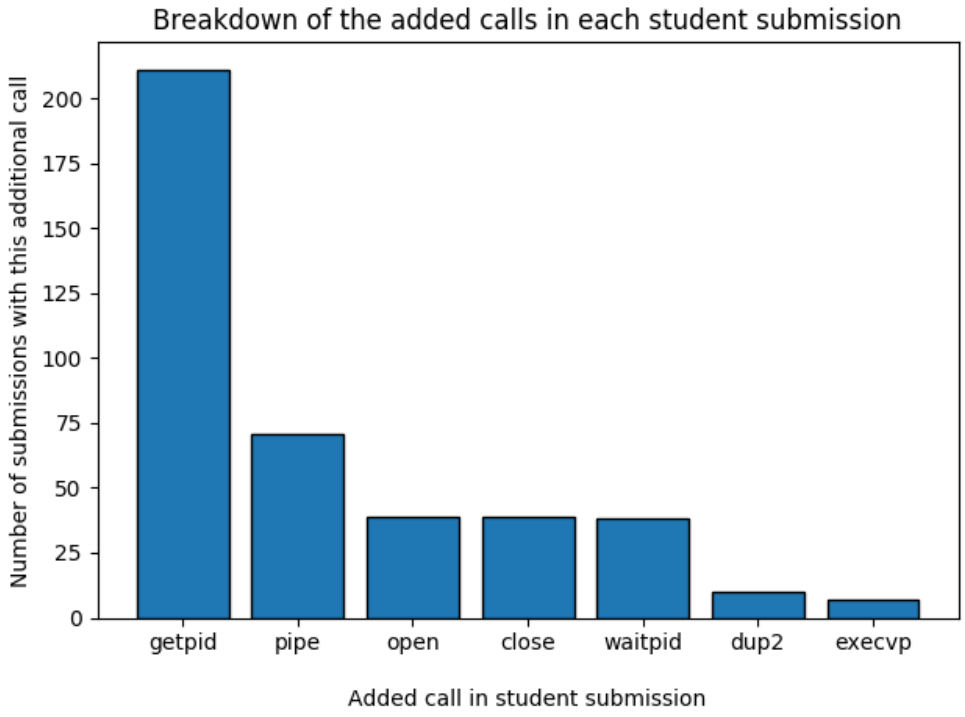

Figure 5.7: Breakdown of the Unnecessary Call type of ShellTrace errors

EshMD child assignment error type. That error class deals with redundant writes in the child process. The exact error is trying to assign the *pid* of the process being executed to a field in the *esh_command* struct when the child process is about to call *exec* shown in Figure 5.8a. The assignment will not be present in the parent shell which makes use the value. The correct behavior would be to assign it in the parent shown in Figure 5.8b which does not require a call to *getpid*. While ShellTrace does identify this error, it displays the error as an unnecessary call to *getpid* instead of the actual cause — an assignment with no effect in the child process.

For the Missing Call error type, we saw expected results. The breakdown of the missing calls is shown in Figure 5.9. The missing *close* calls caused the shell to leak file descriptors in either the child or parent process. The missing *waitpid* calls caused the shell to either create zombie processes or to not wait for the execution of a foreground job. The missing *setpgid*

(a) Student code erroneously assigning *pid* in the child

```
1 if((pid = fork()) == 0){
2   // child process
3   command->pid = getpid();
4   ...
5   execvp(...)
6 }
7 //parent process
```

(b) The correct method of assigning *pid* to the *esh_command*

```
1 if((pid = fork()) == 0){
2   // child process
3   ...
4   execvp(...)
5 }
6 //parent process
7 command->pid = pid;
```

Figure 5.8: Student error resulting in an unnecessary call to *getpid*

calls caused the process groups not to be set up correctly, which would also be flagged by the EshMD tool. The missing *open* calls resulted from the student's improper implementation of I/O redirection where the file to redirect must be *open*ed. The missing *pipe* calls occurred in solutions that did not call *pipe* the appropriate number of times due to either not supporting piping or having an off by one in the piping logic. The missing *exec* calls resulted from a crash in between the *fork* and *exec* of the child.

Students were asked in the post-survey to identify the errors that ShellTrace flagged in their submissions. The errors identified were manually categorized into the ShellTrace error types. Figure 5.10 shows the resulting distribution. We see that most students found that the tools reported they were performing unnecessary actions such as an additional *getpid* call.

5.2.2 Effectiveness of ShellTrace

ShellTrace had less positive feedback than EshMD in the student survey. Figure 5.11 shows the student responses about the perceived effectiveness of ShellTrace.

The main feedback that students gave was that the output was hard to understand and that there were other bugs present in their solution that they wished the tool had flagged first.

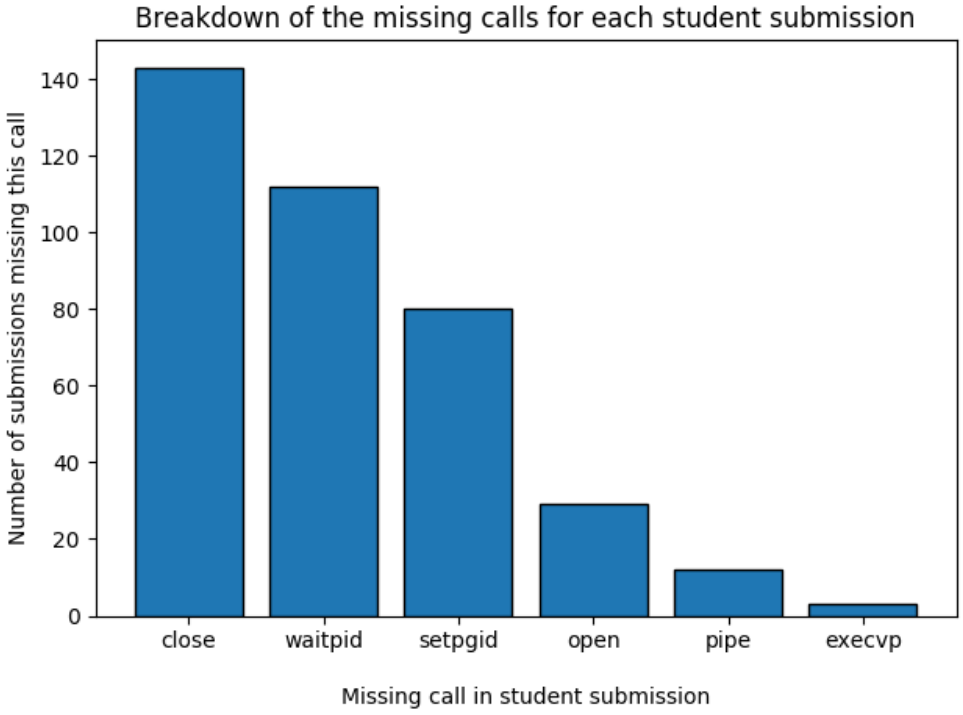

Figure 5.9: Breakdown of the Missing Call type of ShellTrace errors

Looking at the bugs that were reported by ShellTrace as well as the feedback students gave, it appears that ShellTrace is most effective when the program crashes (which are flagged as fork mismatch) have been fixed.

ShellTrace could be used with an existing functional implementation to identify extraneous calls that cause waste. In the first reference implementation that we prototyped for the shell project, we were able to identify an unnecessary *pipe* call and corresponding *close* calls by looking at the resulting graph. While this *pipe* call did not affect the correctness of the shell with respect to the tests, it would be wasteful in terms of the resource use (having more file descriptors open at a given time) and computation time (having to allocate a buffer for the pipe that is never used). A subsequent evaluation should look at how ShellTrace is useful to experts who want to evaluate if their project behaves as intended.

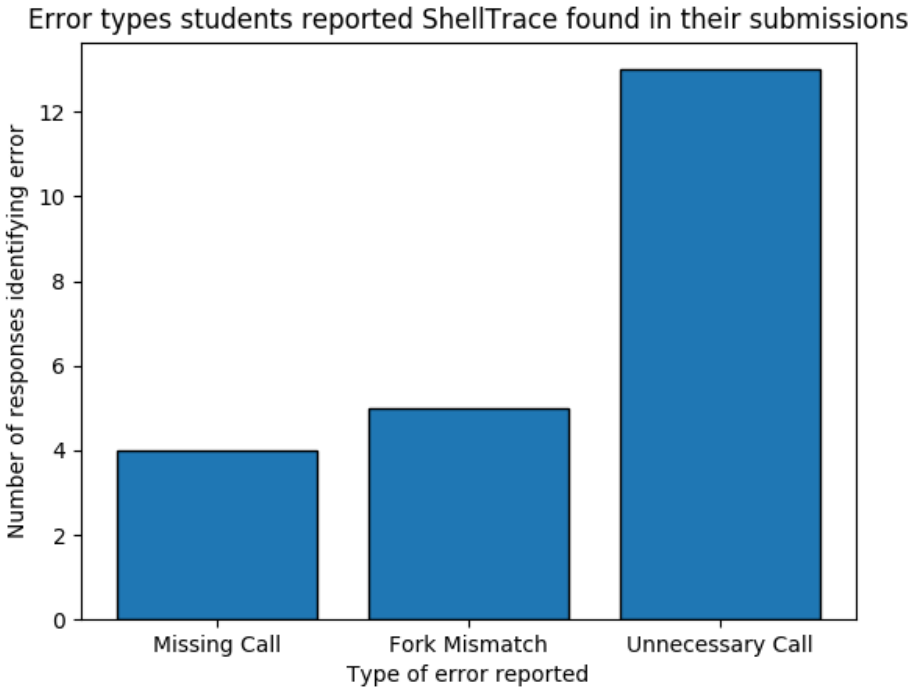

Figure 5.10: Error types students identified as flagged by ShellTrace

5.3 Summary

In our evaluation of the analysis tools EshMD and ShellTrace, we found many bugs in student solutions. The EshMD tool, in particular, was able to point directly at the root cause of student errors. A significant number of students also made a second submission to the analysis tools, showing that they were actively using the tools to help understand why their code was failing the provided tests. We saw that over 50% of the survey responses found both tools useful.

We also saw improvements that can be made to our tools, including the need for better error messages that are more tailored to the project requirements. While a tool like ShellTrace can be used to reason about the overall execution of a program, using the targeted checkers of EshMD is important to point first to specific errors that may cause confusing output in

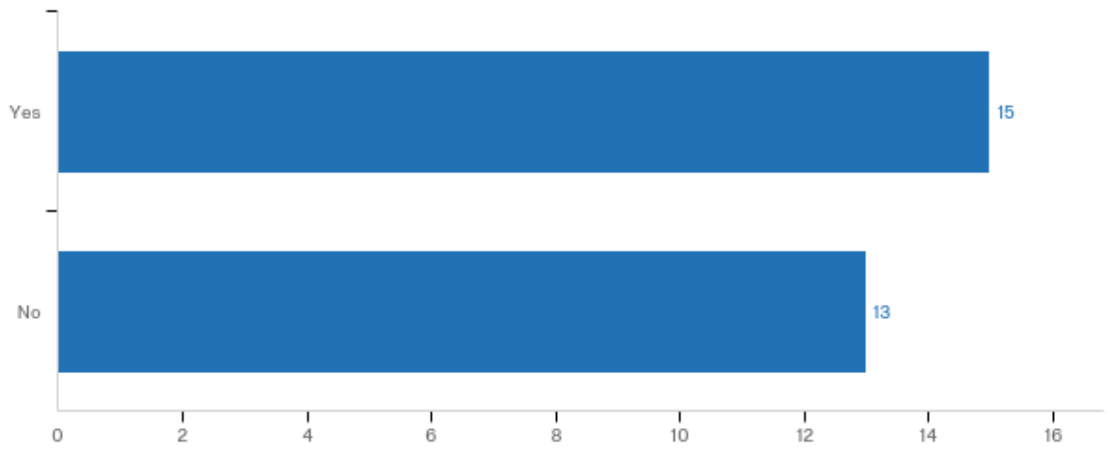

Figure 5.11: Survey results for "Did you find ShellTrace helpful in detecting errors?"

ShellTrace.

Chapter 6

Related Work

6.1 Analysis Tools

This section details a variety of analysis tools and techniques. We highlight how they are similar to this work and how our work differs.

6.1.1 Dataflow Analysis

For our static analysis, we chose to use the Clang Static Analyzer [10] which performs symbolic execution. Symbolic execution is not the only method for static analysis. Dataflow analysis [27] computes properties of the program using its control flow graph. Unlike symbolic execution, dataflow analysis does not explore every possible program state, but rather computes conservative approximations of program properties which can be computed efficiently. For instance, dataflow analysis is used by compilers to determine the reachability of variables and blocks of code. A version of dataflow analysis is used by the security community in the form of taint analysis [35] that determines which statements can be influenced by potentially tainted user input. This analysis allows users to identify sections of their code that are potentially exploitable.

6.1.2 Aspect-Oriented Programming

Aspect Oriented Programming (AOP) is a programming paradigm that separates out cross-cutting concerns (such as logging) from the specific points in code they are implemented [26]. AOP defines cut points which allow the specification of hooks into the code. When the hooks are reached, instrumentation code is added to address the cross-cutting concern. The most used Aspect Oriented Programming implementation is AspectJ [40] for the Java language, although there exist aspect oriented implementations for other languages such as AspectC [12] for the C language. The implementation of our two analysis tools involves instrumentation points that are similar to the AOP cut points, where we specify a point in code such as write instructions, function invocations, and function returns.

6.1.3 Binary Instrumentation

While our tools operate at the source level and the intermediate compiler representation, there are also tools that work on the binary level, performing dynamic binary instrumentation (DBI). DBI adds in analysis code to the program under analysis at runtime. This is typically performed by using a process known as dynamic binary re-compilation where the executable's instructions are loaded by the instrumentation tool, recompiled with instrumentation code, then run. By operating at the binary level, these tools can operate on any arbitrary executable even when the source code is not available. There are several widely used DBI frameworks which allow developers to write custom analysis tools including Valgrind [34], Pin [32], and DynamicRio [8].

For our analysis, chose to use LLVM because the student submissions contain the C source code. The use of LLVM also reduces the slowdown of the program at runtime relative to DBI as the instrumentation can be performed at compile time before optimizations are applied

instead of requiring the code to be instrumented at runtime.

LLVM has been used as a basis for tools used in industry. The Address Sanitizer [36] detects misuses of memory address spaces such as writing off the end of an allocated array. The Thread Sanitizer [37] finds data races in a program.

6.2 Computer Science Education

This section highlights previous work in the area of computer science education regarding tools that aid in student learning.

6.2.1 Automated Feedback

Automated feedback has been studied in a variety of contexts related to computer science education. Web-CAT provides feedback on the test coverage attained by the test cases students write themselves to encourage students to write their own test cases [16]. Halderman et al. take a set of tests, error categories, and debugging hints to provide feedback to students [22]. They use machine learning to classify errors into categories and give the students the proper hint for that category as feedback. Our tools directly flag the errors and provide hints about the root causes.

Static analysis has also been used to judge the code quality of student programs as well as identify the structure of the solution and compare it to the instructor's intended solution [43]. Our tools look at the functionality of the submissions for comparison, rather than the quality of code.

6.2.2 Enhancing Error Messages

Both EshMD and ShellTrace embed domain-specific knowledge about the nature of bugs in the shell project to provide error messages that point to the root cause of the bug. Related work has been done to provide better error messages by enhancing the error messages a compiler provides and evaluating the effectiveness of these enhanced error messages on novice programmers. Denny et al. found that the enhanced compiler error messages did not have a significant effect on the novice student's debugging process [14]. However, Becker found that enhanced compiler error messages reduced the overall error rate in novice programmers [5].

6.2.3 Visualization Tools

Visualization tools have been developed to aid in student understanding from the first programming class a student will take to advanced operating systems classes. There are many tools that visualize C programs by including the current state of the stack and heap at a given line in the program execution [17, 21, 25, 39]. ShellTrace does not target the novice programmer and instead looks to abstract away the details of the implementation and visualize the calls of interest that the implementation performs.

Giraldeau et al. visualize the time user programs spend in system calls and executing on the CPU [20]. They used this to teach operating systems concepts such as scheduling by having students write user programs and reflect on the observed execution.

Chapter 7

Future Work

Our evaluation showed that ShellTrace and EshMD were useful to students. However, there is work to be done to improve them which includes:

1. *Improving the explanation given for ShellTrace errors.* While ShellTrace was able to detect errors in the student submissions, many students did not understand what missing a call or having an extra call in the generated graph meant for their solution or how to interpret the graph. ShellTrace could be improved by pointing the nodes back to the places in the source code that generated them and providing the mapping between the trace that ShellTrace observed and the nodes in the graph.

2. *Expanding the extensible shell project test suite.* Our initial study showed that file descriptor related bugs were present in a large portion of student submissions to EshMD. We could add specific tests to the project test suite that directly flag these errors. These tests could consist of executing special programs that check the file descriptors they have open when they are executed and flag when there are too many or too few file descriptors open.

3. *Running further experiments on the educational value of the tools.* The current evaluation showed that students viewed the tools useful and the tools were able to detect bugs. However, the evaluation lacked a study to determine if student understanding of OS concepts improved with use of the tools. A future study should incorporate an

assessment whether students' understanding increases through the use of our tools.

Chapter 8

Conclusion

An operating systems course is essential to many undergraduate computer science curricula. To give a better understanding of OS concepts students are asked to implement projects that make use of OS abstractions. Implementing projects provides both practical and theoretical knowledge about the concepts as well as how they can be used to build software. Students must possess a deep understanding of the abstractions to implement the programming projects successfully. Misunderstanding about the abstractions provided lead to implementation errors.

In our work, we developed analysis tools to address common bugs that students made in their implementation of the shell project. We created EshMD, which uses both dynamic and static analysis to flag issues with the use of the OS system calls. We also created ShellTrace which derives the project specification from the calls in a reference solution to identify divergent behavior in the student solution. These tools could be extended to other projects that make use of OS abstractions. Finally, we evaluated the usefulness of the tools as perceived by students and number of bugs detected in submissions during a pilot study. The feedback was positive, showing that these tools can be useful in helping students identify and understand the bugs that they make in their implementations.

Bibliography

[1] "clang" C Language Family Frontend for LLVM. https://clang.llvm.org/, 2007.

[2] LLVM Language Reference Manual LLVM 7 documentation. https://llvm.org/docs/LangRef.html, 2018.

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 47–65, 2000.

[4] László Babai and Eugene M. Luks. Canonical labeling of graphs. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 171–183, 1983.

[5] Brett A. Becker. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, SIGCSE '16, pages 126–131, 2016.

[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, February 2010.

[7] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT– a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, 1975.

[8] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, 2003.

[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, pages 322–335, 2006.

[10] Clang Static Analyzer. http://clang-analyzer.llvm.org/, 2016.

[11] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, May 1976.

[12] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectC to Improve the Modularity of Path-specific Customization in Operating System Code. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 88–98, 2001.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[14] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, ITiCSE '14, pages 273–278, 2014.

[15] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 270–280, 2008.

[16] Stephen H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 26–30, 2004.

[17] Matthew Heinsen Egan and Chris McDonald. Program Visualization and Explanation for Novice C Programmers. In *Proceedings of the Sixteenth Australasian Computing Education Conference*, volume 148 of *ACE '14*, pages 51–57, 2014.

[18] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: Concurrency Analysis for Software-defined Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 402–415, 2016.

[19] Free Software Foundation. GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/, 2018.

[20] Francis Giraldeau and Polytechnique Montreal. Teaching Operating Systems Concepts with Execution Visualization. *2014 ASEE Annual Conference & Exposition*, pages 1–24, June 2010.

[21] Philip J. Guo. Online Python Tutor: Embeddable Web-based Program Visualization for CS Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, 2013.

[22] Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. Providing meaningful feedback for autograd-

ing of programming assignments. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 278–283, 2018.

[23] Randolph Herber. Defunct, zombie and immortal processes. https://www-cdf.fnal.gov/offline/UNIX_Concepts/concepts.zombies.txt, 1997.

[24] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira, Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, 2014.

[25] Ryosuke Ishizue, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. PVC: Visualizing C Programs on Web Browsers for Novices. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, pages 245–250, 2018.

[26] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, pages 220–242, 1997.

[27] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, 1973.

[28] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[29] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* http://llvm.cs.uiuc.edu.

[30] Chris Lattner. *Macroscopic Data Structure Analysis and Optimization*. PhD thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, May 2005. *See* http://llvm.cs.uiuc.edu.

[31] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–87, 2004.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[33] Michael S Mahoney. The unix oral history project. https://www.princeton.edu/~hos/Mahoney/expotape.htm, 1989.

[34] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007.

[35] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331, 2010.

[36] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC'12, pages 28–38, 2012.

[37] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, 2009.

[38] E. Stepanov and K. Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, Feb 2015.

[39] Jaishankar Sundararaman and Godmar Back. HDPV: Interactive, Faithful, In-vivo Runtime State Visualization for C/C++ and Java. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pages 47–56, 2008.

[40] The Eclipse Foundation. The AspectJ Project. http://www.eclipse.org/aspectj/, 2013.

[41] The MITRE Corporation. CWE-364: Signal Handler Race Condition. https://cwe.mitre.org/data/definitions/364.html, 2018.

[42] The MITRE Corporation. CWE-403: Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak'). https://cwe.mitre.org/data/definitions/403.html, 2018.

[43] Nghi Truong, Paul Roe, and Peter Bancroft. Static Analysis of Students' Java Programs. In *Proceedings of the Sixth Australasian Conference on Computing Education*, volume 30 of *ACE '04*, pages 317–325, 2004.