UNIVERSITY OF CALIFORNIA
RIVERSIDE

Dynamic Analysis Techniques for Effective and Efficient Debugging

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Yan Wang

August 2014

Dissertation Committee:

Dr. Rajiv Gupta , Co-Chairperson
Dr. Iulian Neamtiu, Co-Chairperson
Dr. Harsha V. Madhyastha
Dr. Zizhong Chen

The Dissertation of Yan Wang is approved:

Committee Co-Chairperson

Committee Co-Chairperson

University of California, Riverside

Acknowledgments

This dissertation would not have been possible without all people who have helped me during my Ph.D. study and my life.

First, I am deeply indebted to my advisor, Dr. Rajiv Gupta, for supporting me throughout the past five years. His patience, encouragement, and belief in my potential during the difficult times when I did not believe in myself make this dissertation all possible. I cannot thank him enough for his step by step mentoring from finding and solving research problems to writing and presenting new ideas. I will never forget he is the person that went over my papers tens of times with me for accuracy. Eventually all his efforts make my doctoral study in UC Riverside the most rewarding and memorable experience that I will cherish forever. *Thanks, Dr. Gupta!*

I am also deeply indebted to my co-advisor Dr. Iulian Neamtiu for his efforts in helping me do good research. I will definitely remember forever that he is the person that stayed up at 5 am with me several times for paper deadlines. His attitude towards research has influenced me immensely in the past and in the future. *Thanks, Dr. Neamtiu!*

Next, I would like to thank the members of my dissertation committee Dr. Harsha V. Madhyastha and Dr. Zizhong Chen for being always supportive. Their constructive comments make this dissertation much better. I also thank Dr. Laxmi N. Bhuyan for his useful courses and help towards this dissertation.

I am also thankful for all my collaborators Dr. Harish Patil, Dr. Cristiano Pereira and Dr. Gregory Lueck at Intel. I am so lucky to work with these knowledgeable and excellent scholars. The collaboration with them is really valuable and has greatly improved this dissertation.

I would like to express my gratitude to all my lab-mates: Dennis Jeffrey, Vijay Nagarajan, Chen Tian, Min Feng, Kishore K. Pusukuri, Changhui Lin, Sai Charan Koduru, Bo Zhou, Vineet Singh, Amlan Kusum, Farzad Khorasani, Keval Vora, Pamela Bhattacharya, Tanzirul Azim, YongJian Hu, Mehmet Esat Belviranli and Sambit Shukla for helping me in many ways during my graduate study. You are priceless friends in my life. Especially, I want to thank Min Feng and Changhui Lin for supporting and encouraging me through the difficult times during my Ph.D. study.

I also want to thank my dear friends Yingyi, Zhixing, Yuan, Yanping, Sheng, Bing, Le, Xiaojing, Ergude, Dung, Yi, Yang, Li, Yuanqi and Lijuan for helping and enriching my life in Riverside and in the future.

Finally, I would like to thank my family. Their endless love and encouragement help me overcome all kinds of difficulties I encountered during my study and my life. I love you forever!

To my family and my love.

ABSTRACT OF THE DISSERTATION

Dynamic Analysis Techniques for Effective and Efficient Debugging

by

Yan Wang

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, August 2014
Dr. Rajiv Gupta , Co-Chairperson
Dr. Iulian Neamtiu, Co-Chairperson

Debugging is a tedious and time-consuming part of software development. Therefore, providing effective and efficient debugging tools is essential for improving programmers' productivity. Existing tools for debugging suffer from various drawbacks – general-purpose debuggers provide little guidance for the programmers in locating the bug source, while specialized debuggers require knowledge of the type of bug encountered. This dissertation makes several advances in debugging leading to an effective, efficient, and extensible framework for interactive debugging of singlethreaded programs and deterministic debugging of multithreaded programs.

This dissertation presents the `Qzdb` debugger for singlethreaded programs that raises the abstraction level of debugging by introducing high-level and powerful *state alteration* and *state inspection* capabilities. Case studies on real reported bugs in popular programs demonstrate its effectiveness. To support integration of specialized debugging algorithms into `Qzdb`, a new approach for constructing debuggers is developed that employs declarative specification of bug conditions and their root causes, and automatic generation of debugger code. Experiments show that about 3,300 lines of C code are generated automatically from only 8 lines of specification for 6 types of memory bugs.

Thanks to the effective generated bug locators, for the 8 real-worlds bugs we have applied our approach to, users have to examine just 1 to 16 instructions. To reduce the runtime overhead of dynamic analysis used during debugging, *relevant input analysis* is developed and employed to carry out input simplification and execution simplification which reduce the length of analysed execution by reducing the input size and limiting the analysis to subset of the execution. Experiments show that the *relevant input analysis* algorithm for input simplification is both efficient and effective – it only requires 11% to 21% test runs of that needed by the standard delta debugging algorithm and generates even smaller inputs.

Finally, to demonstrate that the above approach can also be used for debugging multithreaded programs, this dissertation presents `DrDebug`, a deterministic and cyclic debugging framework. `DrDebug` allows efficient debugging by tailoring the scope of replay to a buggy *execution region* and an *execution slice* of a buggy region. Case studies of real reported concurrency bugs show that the buggy *execution region* size is less than 1 million instructions and the lengths of buggy *execution region* and *execution slice* are less than 15% and 7% of the total execution respectively.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Debugging is a tedious and time-consuming process for software developers. It has been observed that debugging-related tasks (i.e., locating bugs and correcting programs) can take up to 70% of the total time expended on software development and maintenance [51]. Therefore, providing effective and efficient debugging tools is essential for improving developers' productivity.

Extensive research has been conducted on debugging with the goal of developing debugging techniques and tools with diverse capabilities. General-purpose debuggers [26, 13] are in wide use, but they only provide low-level commands, and do not sufficiently guide the programmer in narrowing the source of error; hence even with their use, the task of debugging remains very tedious. Extending the capabilities of such debuggers is also a daunting task. Another class of (more specialized) debuggers are tools that have been designed to detect the presence of specific kinds of bugs such as buffer overflows [18], dangling pointer dereferences [19], memory leaks [108] etc. However, programmers must first know the kind of bug present in the program to make use of such specialized debuggers. Moreover, when faulty code is encountered during execution, its impact on program execution might be observed much later, making it hard to locate the faulty code. Therefore programmers may still need to resort to general-purpose debuggers to understand and fix the bug. To address the above drawbacks, this dissertation presents a debugging framework with following characteristics:

(i) Powerful Dynamic Analysis Framework. The debugger is built around a powerful dynamic analysis framework based upon dynamic binary instrumentation. This allows incorporation of complex dynamic analyses needed to implement high-level commands as well as simple dynamic analyses needed for implementing low-level commands. The high-level commands allow the programmer to progressively narrow the bug to smaller regions of the code and the low-level commands can be employed to understand the detailed behavior of small section of code containing the bug. The debugger is extensible, because a new command can be added by simply extending the user interface and implementing the dynamic analysis needed to implement the new command.

(ii) Integrating Specialized Debugging Techniques. To enable integration of specialized debugging techniques with ease, a bug specification language is provided. The user specifies new types of bugs[1] using simple specifications and the code that performs the dynamic analysis needed for the detection of the specified bugs is then automatically generated. Once again, the powerful dynamic analysis framework on which the debugger is based enables such extensibility to be supported in a systematic fashion.

(iii) Runtime Efficiency. Since powerful dynamic analysis algorithms can incur high runtime execution cost, a two-pronged approach is employed to speed up their use

[1]We define a *bug* to be a class of faults, for example, a double-free bug refers to all double-free faults present in the program.

during cyclic debugging. First, checkpoint and rollback mechanisms are supported, to allow cyclic debugging to be performed without having to repeatedly execute the program from the start – the execution region of interest to the programmer can be rerun multiple times. Second, a complementary approach that simplifies a failing input, and its corresponding execution, is employed such that the length of program execution over which dynamic analysis is performed can be greatly reduced.

The first part of this dissertation presents the `Qzdb` debugger for singlethreaded applications which embodies the characteristics described above. It supports a wide range state inspection and state alteration capabilities that help the programmer to quickly narrow the bug to a small section of code. The second part of this dissertation demonstrates that the above capabilities can also be transferred to tools for debugging multithreaded applications by presenting the `DrDebug` debugger. `DrDebug` relies upon the use of a record and replay component so that relevant parts of program execution can be replayed deterministically. The remainder of this chapter provides an overview of the debugging capabilities of `Qzdb` and `DrDebug` and the organization of the remainder of the thesis.

1.1 `Qzdb`: Interactive Debugger for Singlethreaded Programs

The overview of the `Qzdb` debugger is shown in Figure 1.1. This debugger consists of the debugging interface via which the programmer can debug an execution using a wide range of high-level and low-level commands. The interface communicates with the dynamic analysis framework that implements the corresponding dynamic analyses.

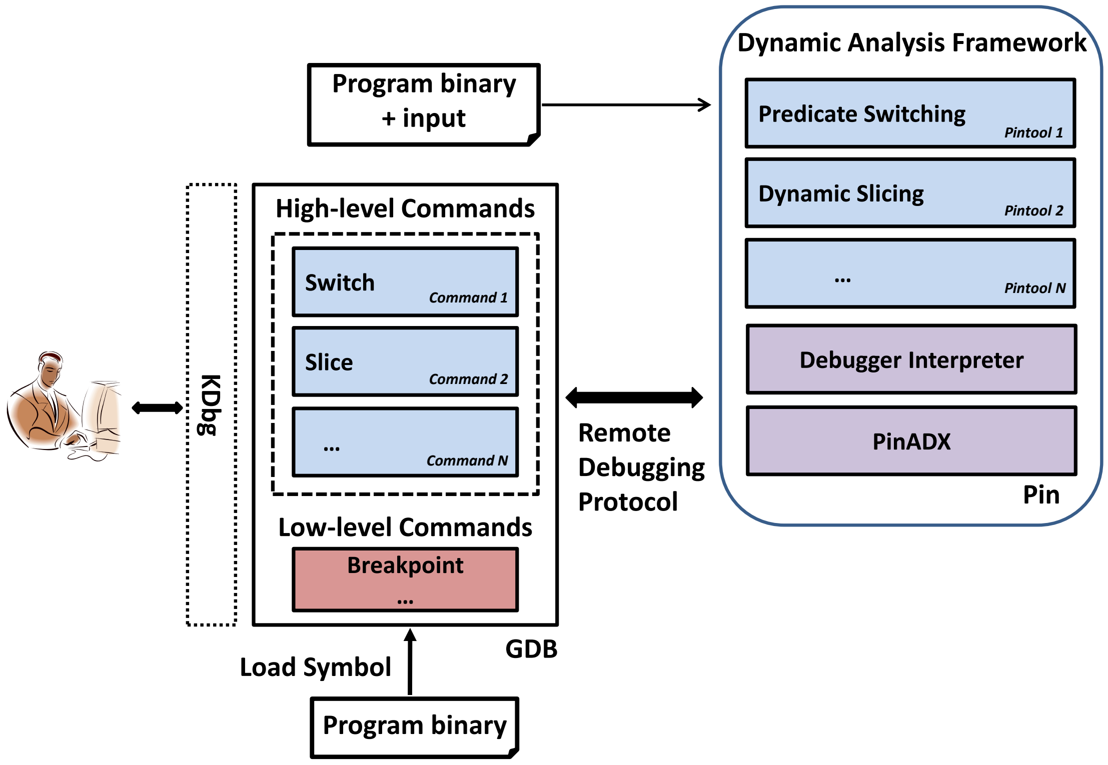


Figure 1.1: The `Qzdg` debugger.

1.1.1 Debugging via *State Alteration & State Inspection*

The `Qzdb` debugger supports powerful, high-level *state alteration* and *state inspection* capabilities to raise the level of abstraction of debugging. The *state alteration* commands dynamically switch the directions of conditional branches or suppress the execution of statements, allowing programmers to narrow faulty code down to a function. The *state inspection* commands allow efficient examination of large code regions by navigating and pruning *dynamic slices* and zooming-in on chains of dynamic dependences. Finally, the programmer can zoom to a small set of statements in a slice by breakpointing at those statements and examining program state.

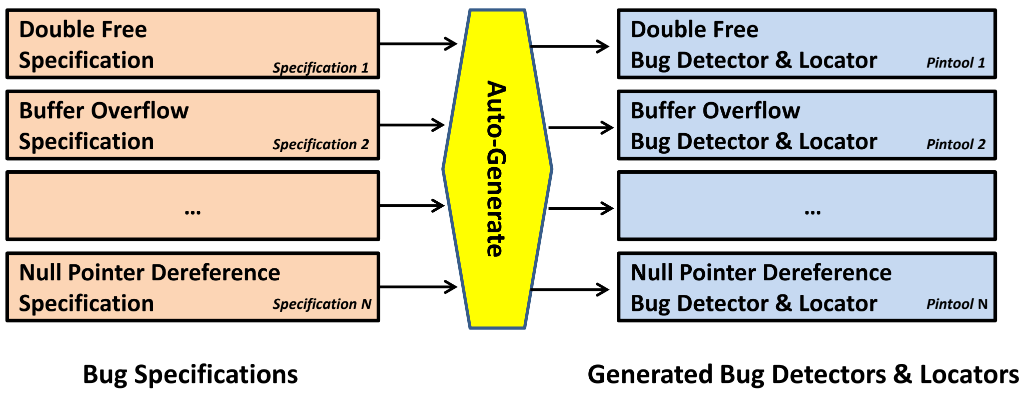

Double Free Specification
Specification 1
Buffer Overflow Specification
Specification 2
...
Null Pointer Dereference Specification
Specification N
Auto-Generate
Double Free Bug Detector & Locator
Pintool 1
Buffer Overflow Bug Detector & Locator
Pintool 2
...
Null Pointer Dereference Bug Detector & Locator
Pintool N
Bug Specifications
Generated Bug Detectors & Locators

Figure 1.2: Extensibility via *bug specification & debugger generation.*

1.1.2 Extensibility via *Bug Specification & Debugger Generation*

This dissertation makes `Qzdb` extensible by presenting a new approach for constructing debuggers based upon: declarative specification of bug conditions and their root causes; and automatic generation of dynamic analysis required to perform bug detection and location. For each bug class, bug conditions and their root causes are specified declaratively, in first-order logic. As shown in Figure 1.2, this approach is demonstrated by extending `Qzdb` with the capability to detect memory bugs such as double free, buffer overflow, and null pointer dereference. Besides, to facilitate locating bugs, the new concept of value propagation chains is introduced to reduce programmers' burden by narrowing the fault to a handful of executed instructions. Finally, the debugging interface is extended to allow the programmer to turn on and off the detection and location of different kinds of bugs.

1.1.3 Improved Efficiency via *Failing Input & Execution Simplification*

This dissertation reduces the overhead of `Qzdb` by simplification of failing *program input* and its *dynamic execution* using a technique we named *relevant input analysis.* The overhead of dynamic analysis required for high-level commands (i.e., predicate

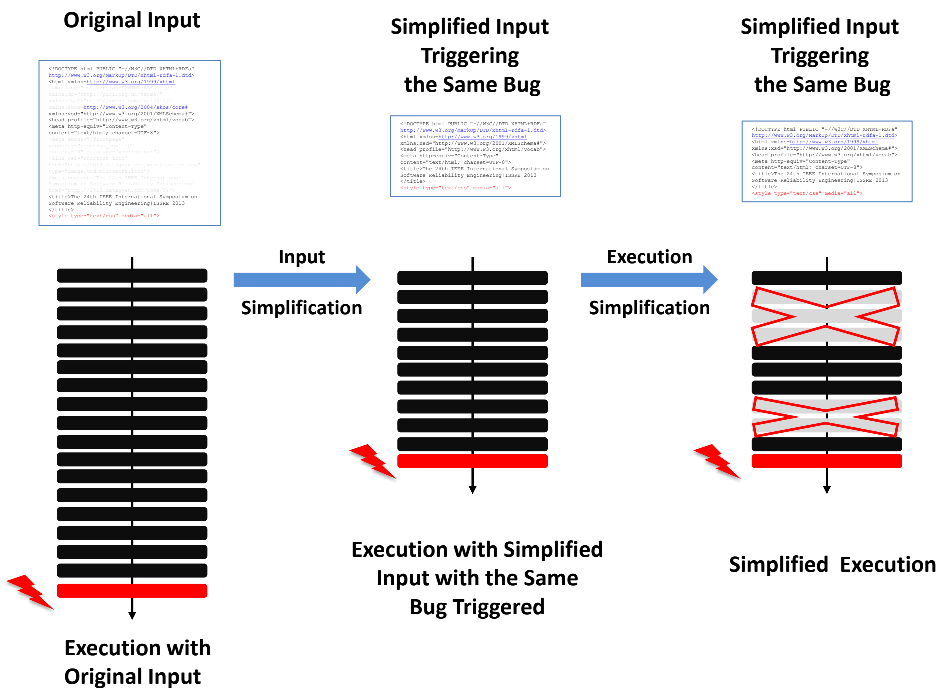

Original Input
Simplified Input Triggering the Same Bug
Simplified Input Triggering the Same Bug
Input Simplification
Execution Simplification
Execution with Original Input
Execution with Simplified Input with the Same Bug Triggered
Simplified Execution

Figure 1.3: Improved efficiency via *failing input & execution simplification*.

switching, dynamic slicing) and automatically generated debuggers (i.e., generated double free bug detector and locator) can be quite high, as extensive runtime monitoring is needed for implementing them. This is particularly the case for long program executions. This dissertation tackles this problem by presenting *relevant input analysis* which is used to accelerate the delta debugging [112, 111, 70] algorithm for *input simplification* while guaranteeing that the same failure manifests on the simplified input. In addition, *execution simplication* is employed to skip parts of execution that are irrelevant to the encountered bug. Figure 1.3 gives an overview of the impact of input simplification and execution simplification.

In summary, `Qzdb`'s novel capabilities assist the programmer during debugging in the following ways. First, the programmer simplifies a failing input as well as its dynamic execution, and then starts debugging using the simplified execution. Second, the programmer enjoys the advantages of many specialized yet auto-generated bug check-

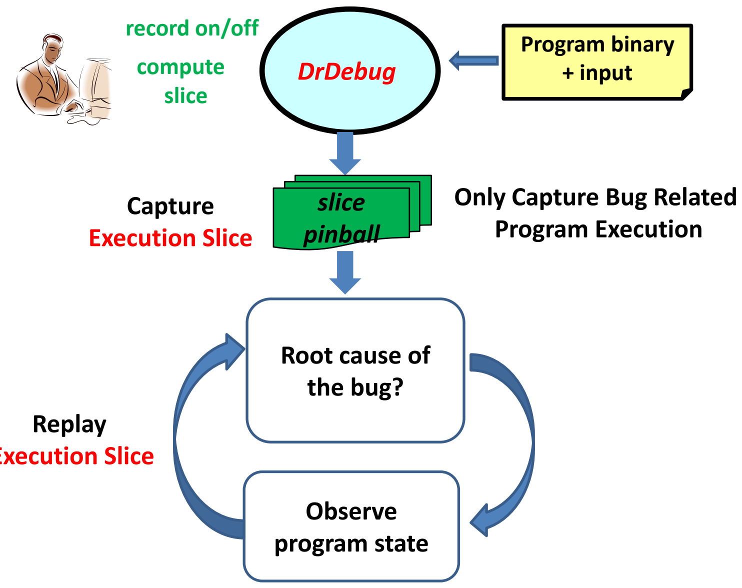


Figure 1.4: `DrDebug`: interactive debugger for multithreaded programs.

ers with negligible programming effort while benefiting from value propagation chains that perform root cause localization. Third, for complicated bugs, the programmer can leverage the high-level *state alteration* and *state inspection* capabilities to speed up bug fixing.

1.2 `DrDebug`: Interactive Debugger for Multithreaded Programs

With the advent of multicores, programmers must write parallel programs to achieve increased performance. However, writing and debugging multithreaded programs is very difficult; hence this dissertation extends the above debugging framework to multithreaded programs. Cyclic debugging for multithreaded programs poses multiple challenges: depending on the location of the bug, it can take a very long time to fast-forward and reach the buggy region; heap and stack locations, the outcome of system calls, and thread schedules change between debugging sessions; some bugs are hard to reproduce, in general and also under a debugger. To address these challenges `DrDebug` supports a collection of tools based upon PinPlay, a capture and replay framework.

The overview of the `DrDebug` debugger is shown in Figure 1.4. The features of `DrDebug` significantly increase the efficiency of debugging by tailoring the scope of replay in two ways. First, the scope can be limited to a buggy *execution region* by recording the execution in a region pinball (using a record on/record off capability) and then replaying it during cyclic debugging. Second, while replaying the execution region, the scope can be further narrowed to an *execution slice* of the buggy region by generating a slice pinball and replaying it for debugging. With `DrDebug`, a highly precise dynamic slice is computed that can then be browsed by the user by navigating the dynamic dependence graph with the assistance of our graphical user interface. If the dynamic slice is of interest to the user, it is used to compute an execution slice whose replay can then be carried out efficiently as execution of code segments that do not belong to the execution slice is skipped. `DrDebug` also allows the user to step from the execution of one statement in the slice to the next while examining the values of variables in a live debugging session.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents the commands and the architecture of the `Qzdb` debugger for singlethreaded programs. Chapter 3 shows how `Qzdb` can be extended to include techniques for specific types of bugs – the user provides a declarative specification of a specific kind of bug (conditions and root causes) while the automatic generator processes the specifications and produces the implementations of dynamic analysis for corresponding bug detector and locator. Chapter 4 presents the relevant input analysis technique used to simplify program input as well as its dynamic execution to reduce the runtime overhead of dynamic analysis. Chapter 5 presents `DrDebug` that extends the debugging framework used in `Qzdb` to support debugging of multithreaded programs. Chapter 6 describes the related work. Finally Chapter 7 summarizes the contributions of this dissertation and identifies directions for future work.

Chapter 2

The `Qzdb` Interactive Debugger for Singlethreaded Programs

To assist with debugging, programmers frequently make use of an interactive debugger (e.g., GDB) whose use typically involves: state inspection, state alteration, and code modification. The programmer executes the program on a failing input and uses *state inspection* commands to examine program state at various points (e.g., by setting breakpoints and examining values of variables when breakpoints are encountered). After finding suspicious values, the programmer may apply *state alteration* to correct these values and see how the program's execution is affected. Alternatively, the programmer may perform *code modification* such as commenting out suspicious statements [17] and then recompile and rerun the program, to see how the program behavior is affected. Some debuggers attempt to speed up this process, albeit for restricted scenarios: for example, Visual Studio offers an Edit-and-Continue feature [101] for on-the-fly changes to the program being debugged, but code modifications such as most changes to global or stack data are not supported.

Both state alteration and code modification techniques help the programmer in understanding and locating faulty code. While state alteration is a lightweight technique, code modification slows down debugging as it requires program recompilation and reexecution. This can take a significant amount of time if the program runs for long before exhibiting faulty behavior and the above process is performed repeatedly. A debugger such as GDB supports simple state alteration commands for altering values of variables. Thus the programmers often resort to code modification to locate the bug.

Our `Qzdb` interactive debugger addresses such issues by supporting commands to reduce debugging effort and increase debugging speed. These commands allow the programmer to narrow his/her focus successively to smaller and smaller regions of code. The state alteration commands allow the programmer to narrow faulty code down to a function. The state inspection techniques allow efficient examination of large code regions by navigating and pruning dynamic slices and zooming in on chains of dependences. Finally, the programmer can zoom to a small set of statements in a slice by breakpointing at those statements and examining program state.

The commands in `Qzdb` speed up the iterative debugging process by reducing the need for code modifications, which require recompilation and reexecution. Its state alteration commands allow the programmer to perform control flow alterations by switching outcomes of conditional branches. Its execution suppression commands allow skipping of statements during execution. That is, these commands effectively simulate the effect of code modifications without the need for recompilation. In addition, since `Qzdb` also supports checkpoint and rollback, the programmer can rollback to an earlier execution point, specify state alterations, and then reexecute the program. A reexecution from a checkpoint instead of from the beginning can greatly reduce the waiting time associated with recompilation and reexecution for long executions.

2.1 Debugging Commands

`Qzdb` provides three kinds of debugging commands—**state alteration**, **state inspection**, and **state rollback**—listed in Table 2.1. Of these, six commands—`switch`, `suppress`, `slice`, `record`, `prune`, and `instance` are not found in commonly-used interactive debuggers. Other commands are extended to support new debugging features. State alteration commands are used to isolate bugs and help programmers efficiently gain comprehension of program's faulty behavior. State inspection commands help programmers focus on bug-related statements and present unexpected dependences to the programmers and allow them to navigate along dependence edges. State rollback commands enable the quick reexecution of the suspicious code region. Programmers use these commands via the GDB command line or the GUI.

Summary	**Commands**	**Description**
State Alteration	`switch`	switch the outcome of a predicate
	`suppress`	suppress the execution of a statement
State Inspection	`record`	turn on/off recording for slicing
	`slice`	perform backward dynamic slicing
	`prune`	prune dynamic slices
	`sbreak`	create breakpoints in a slice
	`conditional`	conditionally capture memory bug
	`breakpoint`	related library calls (e.g., *malloc, free*)
	`instance`	print execution instance of a statement
State Rollback	`checkpoint`	set up an incremental checkpoint
	`rollback`	rollback the program state

Table 2.1: Major debugging commands.

`Qzdb` can greatly relieve the burden of programmers. First, when a program crashes, it can be difficult for the programmer to reason about the execution flow, e.g., when the crash happens because a library call destroys an auto-maintained stack or heap. `Qzdb` captures the abnormal data dependences and presents them to the programmers in an intuitive way. Second, it is the programmer's responsibility to find suspicious code and speculate about the root cause. `Qzdb` enables the programmer to focus on bug-related statements and guides the examination of values and setting of breakpoints guided by dynamic slicing. Third, even after discovering all the bug-related statements, the programmer still has to understand and fix the bug. `Qzdb` enables the programmer to quickly identify critical bug manifestation condition (e.g., a critical function invocation) by leveraging state alteration and narrowing the fault to a small code region. Fourth, it is common that a variable use is data-dependent on a far-away definition in a different function or a source file. Navigating across source files is burdensome. `Qzdb` enables the programmer to visually navigate captured dependences and reason about the execution.

2.1.1 State Alteration Interfaces

State alteration commands provide an easy way to alter the program execution state dynamically and enable programmers to avoid repetitive program compilations and executions. `Qzdb` supports state alterations to affect the flow of control and suppress the execution of statements. These features are described next.

2.1.1.1 Switching Control Flow

The command `switch` *file*:*line* [*all*|*once*|*n*] is designed to switch the outcome of the predicate in the *line*th line in file *file*. Programmers can choose to switch the outcome for every (*all*), next (*once*) or only the nth (*n*) instance of the predicate.

Using `switch`, programmers can dynamically change the outcome of a branch and then check the difference in program state and result. When a program crashes or deviates from the desired behavior, programmers use `switch` to invert the outcome of the predicate dynamically. If the program behaves correctly after inversion, the programmer can infer that there is an error in the predicate or the predicate is critical to bug manifestation. Otherwise the predicate is likely unrelated to the error. If there are several predicates in the execution trace of a failing run, all the predicates with which the program works properly by following the inverted branch compose the critical bug manifestation condition. That is, the bug disappears when the outcome of any of these predicates changes, providing valuable clues to the programmer to understand the bug. With the aid of `switch`, programmers avoid source code modification and recompilation which are time-consuming.

2.1.1.2 Execution Suppression

The `suppress` *file*:*line* [*all*|*once*|*n*] command suppresses the execution of statement at line *line* in file *file*. Programmers can choose to suppress every instance (*all*), only the next instance (*once*), or only the nth (n) instance of the statement.

Like `switch`, `suppress` is useful for isolating a bug. A commonly-used debugging strategy is to temporarily comment out a section of code and then check whether the remaining part works as expected [17]. This approach involves recompilation of the source code. The `suppress` command is designed to simplify this procedure. If the programmer suspects that a statement or function is faulty, he can suppress its execution on-the-fly without having to modify the source code, recompile the program and then rerun the program from beginning. For example, assume that the programmer has forgotten to use a guarding predicate around some statements, causing the program to crash. The programmer can use `Qzdb` to suppress the unguarded statements based on his knowledge of the program. If the program then behaves as expected, the programmer can now focus on fixing the code.

Programmers can first suppress a function call to identify a faulty function and then suppress statements in the function to identify faulty code. By reducing the suppression to finer granularities, the root cause of failure can be narrowed to a smaller code section.

2.1.2 State Inspection Interfaces

2.1.2.1 Dynamic Slicing

Dynamic slicing commands include the following.

- `record` *file*:*line* `on`|`off` identifies the code region where dynamic slicing is required.
- `slice` *stmt i variable*|*addr* [*size*]|*register* constructs a backwards dynamic slice for *variable*, memory region [*addr, addr+size*) or *register*, starting from the i^{th} execution instance of *stmt*. If no variable is specified, we generate a slice for all the registers and variables used in current execution instance of *stmt*. Our debugger assigns unique numbers to each generated slice and feeds this number back to the programmer.
- `prune` *id list* is used to prune the id^{th} slice by eliminating from the slice all the dependence edges related to any variable or register in *list*.
- `sbreak` *id* $s_1[, s_2, ...]$ is used to insert a breakpoint at s_1^{th} (and s_2^{th},...) statements in the id^{th} slice. The command `sbreak all` *id* inserts a breakpoint at each statement in the id^{th} slice. Breakpoints for `sbreak` are triggered when specific execution instances are encountered.
- `sdelete` *id* is used to delete the id^{th} slice.
- `info slices` is used to print a detailed report of all generated slices that have not been deleted.
- `instance` *file*:*line* prints the execution instance of $line^{th}$ line in file *file*.

With traditional debuggers, programmers navigate and conjecture the root cause over the whole execution trace [26]. With our debugger, programmers can infer the root cause in the pruned slices of variables with wrong values. These slices are *much smaller*. The `slice` command is very efficient in locating the root causes of bugs. It is common for a failing program to exhibit abnormal control or data dependences that can be quickly identified by examining a `slice` that captures them. For example, for a NULL pointer dereference bug, we can locate where the NULL pointer originates by examining the backwards slice of the NULL pointer. The slice may also help determine if the pointer was mistakenly set to $NULL$.

The `slice` command is also very useful for double free, heap and stack buffer overflow bugs. These memory-related bugs are notoriously hard to find because the source code of library functions is not available and the internal data structures (heap and stack metadata) are transparent to programmers. Our debugger traces into library calls and captures hidden dependences among internal data structures. For example, when a heap buffer overflow bug destroys an internal data structure maintained by the heap allocator, a dependence path from the place where the error is manifested to the overflow point is found. Since library source code is unavailable to programmers, we do not present dependences inside a library code. Instead, we squash the dependence edges to statements inside the library functions to their call sites. For example, consider a stack smashing bug inside a library call that causes a crash when the returning statement is executed. We report a dependence edge from the returning statement to the call site of library function.

During debugging, programmers often have high confidence that the program performs correctly for some execution segments. In that case, they can focus on the most suspicious region first. The `record` command allows recording the concerned code region and slicing based on the partial def-use information. Further, programmers may have high confidence on the correctness of some values. For example, they may know that a loop variable i has nothing to do with the failure. The dynamic slice can be pruned to exclude the dependences due to such values. The `prune` command removes dependence edges corresponding to variables or special registers in *list*. Thus, the `record` and `prune` commands greatly limit slice sizes and save programmers' time and effort.

The `sbreak` command facilitates setting breakpoints efficiently. It generates a breakpoint which is only triggered when the specific execution instance in the slice is reached. Programmers frequently step through the program execution to reason about the control/data flow and find faulty code. With the help of `sbreak all`, the programmer is able to only step through the statements and execution instances in the slice. Because all the statements influencing the value of a variable are included in the slice, stepping only through the statements in the slice reduces programmers' effort.

2.1.2.2 Conditional Breakpoints

Existing debuggers (e.g., GDB) provide conditional breakpoints; however, the condition must be defined at source code level, which is not available to programmers for library functions. Thus, conditional breakpoints must be set at each call site, which is time consuming and inflexible. Therefore we provide a command `breakpoint` *lib_func* [`if` *condition*] that triggers a breakpoint at the call site of *lib_func* when *condition* is satisfied. The condition allows selective and efficient capture of critical library function invocations. The condition has the forms:

- `if arg`*N*|`ret==`*value* triggers a breakpoint when the N^{th} argument or return value equals the given value.

- `if write/read/access` *addr* [*size*] triggers a breakpoint when the function writes/reads/accesses specified memory location.

Extended conditional breakpoints are very useful for memory-related bugs. For example, there may be three possibilities if a program crashes at a *free*— double free, unmatched free (i.e., freeing an unallocated pointer), or heap buffer overflow. The programmer can check for a double-free bug using `breakpoint` *free* `if arg`*1*`==`*fail_addr* to see if this memory region has been freed before. If a previous free with the same address is caught, this indicates a double-free bug. Otherwise, if no previous deallocation is found, programmers can use `breakpoint` *malloc* `if ret==`*fail_addr* to see if crash is due to deallocation of unallocated memory. Programmers can use `breakpoint` *strcpy/memcpy* `if arg`*1*`==`*addr* or `breakpoint` *strcpy/memcpy* `if write` *addr* to find if the specified memory location is modified in *strcpy/memcpy*, to find buffer overflow bugs.

2.1.3 State Rollback Interfaces

The `checkpoint` command creates a checkpoint and the debugger assigns an *id* to it. The command `rollback` *id* is used to go back to a previous checkpoint and re-execute from that point. The command `info checkpoints` prints the list of checkpoints and *cdelete id* deletes a checkpoint.

Traditional checkpointing [50] records/restores memory/register states and is inadequate for us. First, if the programmer rolls back the execution when recording is turned on, the recorded def-use information will wrongly include the rolled-back portion of the execution, thus slices generated based on this information will be incorrect. Second, rolling back the program state to a previous checkpoint will cause inconsistency between the statement execution instances in the previously-generated slice and in the restored program. Our debugger extends the traditional incremental checkpointing mechanism to support state alteration and state inspection. In addition to recording (restoring) the memory and register states, we also record (restore) the execution instances. This extension maintains the consistency between generated slices and program state.

The `checkpoint` and `rollback` commands are particularly useful for iterative debugging. Without state rollback, programmers have to restart the execution every time they go over the possible faulty area or when they want to modify the program execution (e.g., altering input, switching a predicate, or suppressing a function call). Moreover, on systems such as Linux, the addresses of stack- and dynamically-allocated regions vary from run to run due to address space randomization for security. Therefore it is troublesome to diagnose bugs related to dynamically allocated regions (e.g., double free and heap buffer overflow) and stack (e.g., stack smash). Thanks to the `checkpoint` command, programmers can go back to a previous point and rerun the program from there, while keeping all addresses of dynamically allocated regions unchanged. The `rollback` command keeps the addresses the same when programmers rerun the program from a checkpoint.

2.2 Usage of Debugging Commands

This section describes how different types of commands are used during the debugging process and then demonstrates their use in context of a set of hard-to-locate bugs from real programs. Figure 2.1 overviews the debugging process based upon the supported commands. Let us assume that the execution of the program has failed on an input. First, the programmer enters commands that will later allow detailed state inspection and program reexecution. Then the program is executed from the beginning

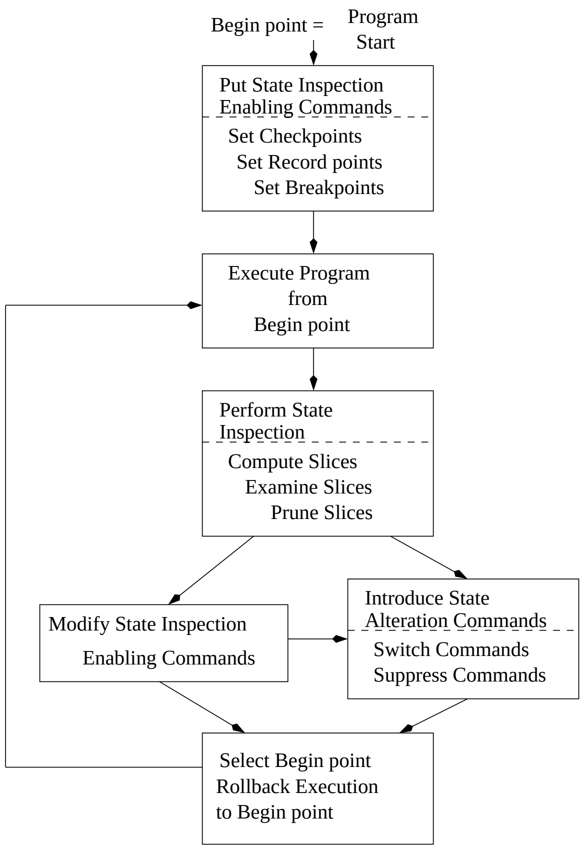
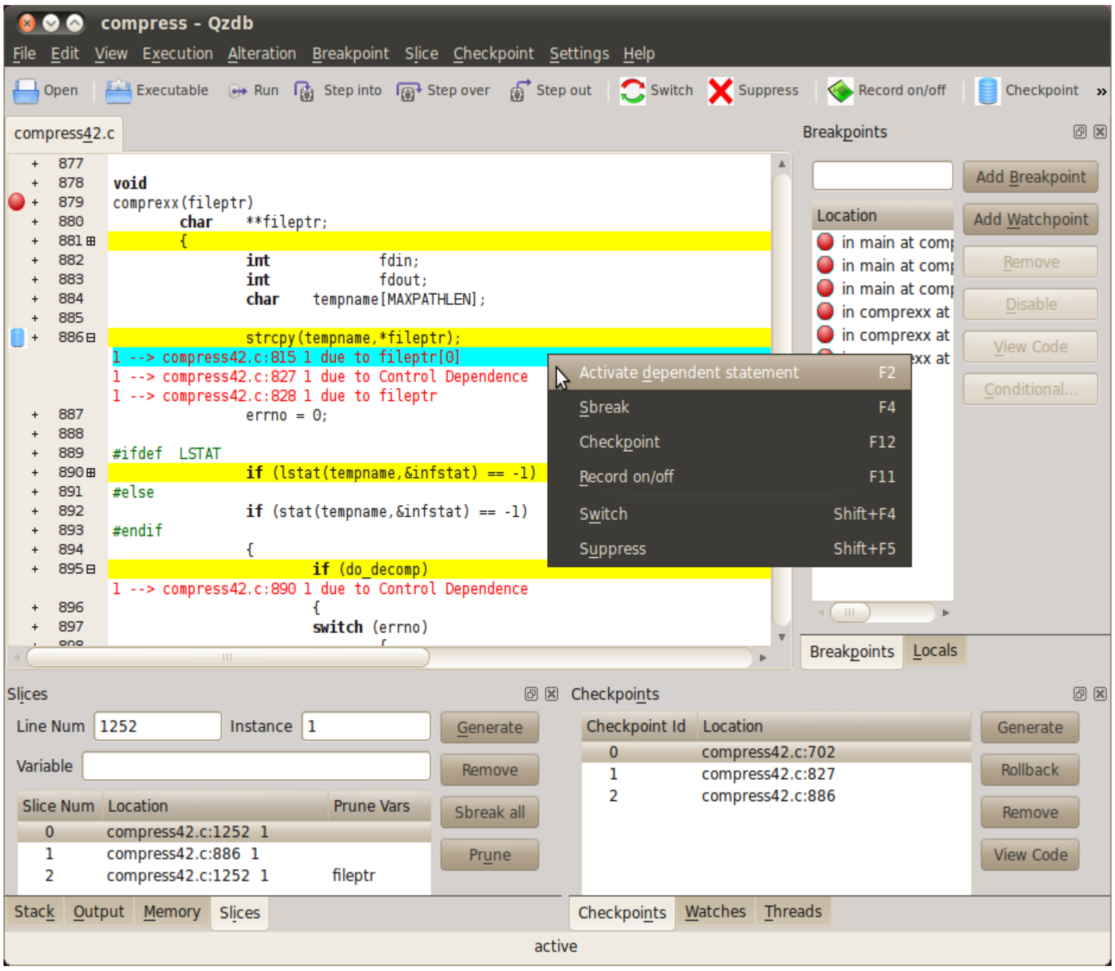

Begin point = Program Start
Put State Inspection Enabling Commands
Set Checkpoints
Set Record points
Set Breakpoints
Execute Program from Begin point
Perform State Inspection
Compute Slices
Examine Slices
Prune Slices
Modify State Inspection Enabling Commands
Introduce State Alteration Commands
Switch Commands
Suppress Commands
Select Begin point
Rollback Execution to Begin point

Figure 2.1: Typical use of our debugger.

until execution stops due to an error or a breakpoint is encountered. The user can now perform state inspection, starting with computing a backward dynamic slice. The programmer can prune the slice based on the knowledge of the program; next, internal execution states can be probed by setting breakpoints at statements in the slice. Based upon the insights gained, the programmer may choose to use state inspection commands and/or apply state alteration techniques to further understand program behavior. By rolling back the execution to an earlier checkpoint, and reexecuting the program from that point, the programmer can observe the impact of state alteration by examining program state. This is an iterative process which eventually leads to location of faulty code. This iterative process does not require program recompilation or reexecution from the beginning.



```
877
878  void
879  comprexx(fileptr)
880          char    **fileptr;
881          {
882                  int             fdin;
883                  int             fdout;
884                  char    tempname[MAXPATHLEN];
885
886                  strcpy(tempname,*fileptr);
     1 --> compress42.c:815 1 due to fileptr[0]
     1 --> compress42.c:827 1 due to Control Dependence
     1 --> compress42.c:828 1 due to fileptr
887                  errno = 0;
888
889  #ifdef  LSTAT
890                  if (lstat(tempname,&infstat) == -1)
891  #else
892                  if (stat(tempname,&infstat) == -1)
893  #endif
894                  {
895                          if (do_decomp)
     1 --> compress42.c:890 1 due to Control Dependence
896                          {
897                          switch (errno)
```

Slice Num	Location	Prune Vars
0	compress42.c:1252 1	
1	compress42.c:886 1	
2	compress42.c:1252 1	fileptr

Checkpoint Id	Location
0	compress42.c:702
1	compress42.c:827
2	compress42.c:886

Figure 2.2: The main window of our debugger.

Next, we present case studies of using `Qzdb`, based upon five different kinds of memory-related bugs listed in Table 2.2, taken from BugNet [76].

2.2.1 Localizing a Stack Smashing Bug

Figure 2.2 shows a stack buffer overflow bug (also referred to as stack smashing) in *ncompress-4.2.4*. Line numbers are shown on the left. The bug is triggered when the length of the input filename (pointed to by *fileptr* at line 880) exceeds the size of array *tempname* defined at line 884 which stores the file name temporarily. The program crashes when the *comprexx* function tries to return to its caller because the return address of *comprexx* is overwritten at line 886 in the *strcpy* function.

Program Name	**LOC**	**Error Type**	**Error Location**
ncompress-4.2.4	1.4K	Stack Smashing	compress42.c:886
Tidy-34132	35.9K	Double Free	istack.c:031
bc-1.06	10.7K	Heap Buffer Overflow	storage.c:176
Ghostscript-8.12	281.0K	Dangling Pointer Use	ttobjs.c:319
Tar-1.13.25	28.4K	NULL Pointer Use	incremen.c:180

Table 2.2: Overview of benchmarks.

Without our debugger it is extremely difficult for programmers to figure out why the program crashes when it executes the *return* statement (at line 946). First, because the program counter is corrupted, existing debuggers (e.g., GDB) cannot report the exact crash point. Our debugger reports the exact location by tracking the modification to the program counter and reporting its current and previous values. Second, the program crashes because a library call destroys the auto-maintained stack, neither of which are visible to the programmer; hence it is difficult to reason about the bug from the source code. Our debugger captures the hidden data dependence and presents it to the programmer.

Returning to our example, with our debugger the programmer knows that the program crashed at line 1252 (shown in Figure 2.3) and the program counter is modified at this crash point. Next, the program can be restarted and additional checkpoints introduced for later use. The programmer can also enable tracing at the beginning of *main* and turn it off at the crash point to later get the whole slice.

With our enhanced dynamic slicing, if the programmer omits variables in the slice criterion, the debugger computes dynamic slices for all registers and variables used in a statement. This is very useful for memory-related bugs because there is no need (evenworse, sometimes it is very difficult) for the programmer to figure out which variables or memory regions are used at the crash point. If we use the statement line number (1252) and instance (1) as the slice criterion, the generated slice is as shown in

Figures 2.2 and 2.3. If the programmer omits the instance, the latest execution instance is used by default. All statements in the slice are highlighted in yellow (e.g., lines 1252 and 886) so programmers can focus on them.

To further help reason about the execution flow, our debugger captures and presents the concrete control/data dependence relationships. Our debugger also allows programmers to navigate the dependence edges and quickly identify unexpected control/data flow. To get the dependence relationships, users click on the left expansion mark of a statement in the slice. The dependence edges from statement $stmt_1$ are shown as follows:

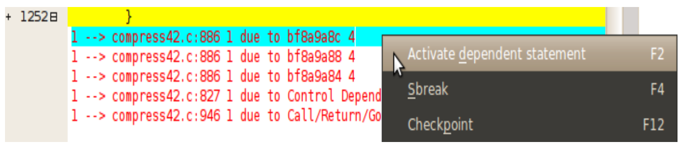
$$instance_1 \rightarrow file_2 : line_2 \ instance_2 \ due \ to \ Memory/ControlDependence$$

which means that the $instance_1^{th}$ execution of $stmt_1$ is data/control dependent on the $instance_2^{th}$ execution of statement at $line_2$ in $file_2$. For example, by clicking the left expansion mark of line 886 in Figure 2.2, all the dependence edges originating from this statement in slice 0 are shown just below the source line (in red). From the first line just below line 886, we can see that its first execution instance is data-dependent on the first execution instance of statement at line 815 due to variable *fileptr[0]*. The programmer can navigate backwards along the dependence edge by clicking the "Activate dependent statement" button (e.g., jump directly to the definition point of *fileptr[0]* at line 815). That is, programmers can navigate backwards from the current statement to the depended statement (predicate in case of control dependence, definition point in case of data dependence) in one click, even when the current program point is in a different function or a different source file from the depended statement. Source code navigation along dependence edges can greatly enhance programmers' debugging efficiency.

Following the dependence edges from the crash point of line 1252, the programmer knows that it is data-dependent on *strcpy* called at line 886 due to an unexpected write access to addresses *0xbf8a9a8c*, *0xbf8a9a88*, and *bf8a9a84* (see the first three dependence edges below line 1252 in Figure 2.3). Experienced programmers will know that there is something wrong with the invocation of *strcpy*. They can rollback the program state to a previous checkpoint, and then use execution suppression to suppress the abnormal data flow and verify that the root cause is *strcpy* invocation.

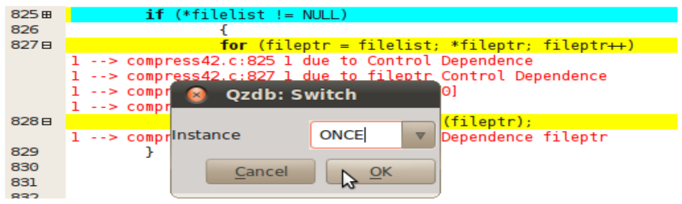
Because the crash point is also control dependent on the statement at line 827 (see fourth dependence edge below line 1252, and line 827 in Figure 2.4), less-experienced programmers may navigate along this dependence edge. If the programmer navigates to line 828, the invocation location of *comprexx*, he can quickly narrow the faulty region by either applying suppression (line 828, Figure 2.5) or predicate switching (lines 825 or 827, Figure 2.4). In both cases the crash goes away. Therefore, the programmer will have high confidence that *comprexx* is faulty. Note that *comprexx* may be invoked multiple times, e.g., when *ncompress* detects multiple files in a folder. Using our debugger, the programmer can easily control which instance to alter. With the help of state alteration, the programmer can quickly zoom into the faulty function *comprexx*. Next, the programmer can rollback to a previous checkpoint and rerun the program up to the beginning of this function, and then efficiently step through *comprexx* with the help of `sbreak all`.

When using slicing, the programmer can use the `prune` command to reduce the size of slices as shown in Figure 2.6. For example, by pruning the slice by *fileptr*, 17% of the original statements in slice 0 are pruned away, and 37% of the dependence edges are eliminated. Programmers can also generate slices limited to function *comprexx* by simply recording just the execution of *comprexx*—doing so reduces the number of statements in



```
        }
1 --> compress42.c:886 1 due to bf8a9a8c 4
1 --> compress42.c:886 1 due to bf8a9a88 4
1 --> compress42.c:886 1 due to bf8a9a84 4
1 --> compress42.c:827 1 due to Control Depend
1 --> compress42.c:946 1 due to Call/Return/Go
```

Figure 2.3: Slicing from the crash point.

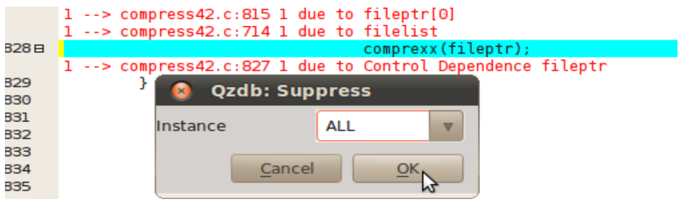


```
        if (*filelist != NULL)
                {
                for (fileptr = filelist; *fileptr; fileptr++)
1 --> compress42.c:825 1 due to Control Dependence
1 --> compress42.c:827 1 due to fileptr Control Dependence
1 --> compr                                      0]
1 --> compr
                                                 (fileptr);
1 --> compr                                      Dependence fileptr
        }
```

Qzdb: Switch

Instance: ONCE

Figure 2.4: Predicate switching.

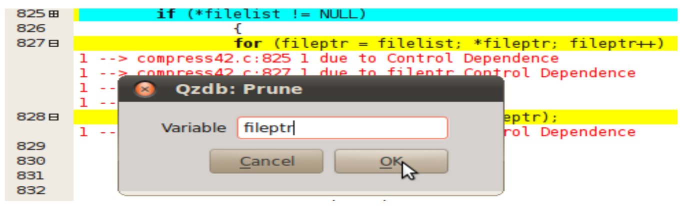


```
1 --> compress42.c:815 1 due to fileptr[0]
1 --> compress42.c:714 1 due to filelist
                                        comprexx(fileptr);
1 --> compress42.c:827 1 due to Control Dependence fileptr
        }
```

Qzdb: Suppress

Instance: ALL

Figure 2.5: Execution suppression.



```
        if (*filelist != NULL)
                {
                for (fileptr = filelist; *fileptr; fileptr++)
1 --> compress42.c:825 1 due to Control Dependence
1 --> compress42.c:827 1 due to fileptr Control Dependence
1 --
1 --
                                              eptr);
1 --                                          rol Dependence
```

Qzdb: Prune

Variable: fileptr

Figure 2.6: Pruning a slice.

```
istack.c:
025: AttVal *DupAttrs( TidyDocImpl* doc, AttVal *attrs) {
033: *newattrs = *attrs;
034: newattrs → next = DupAttrs( doc, attrs → next ); ...
     /* double free due to the missing of the following statement*/
039: newattrs → php=attrs → php? \
       CloneNode(doc, attrs → php):NULL;
041:}
057: void PushInline( TidyDocImpl* doc, Node *node) {...
092: istack → attributes = DupAttrs( doc, node → attributes ); }
097: void PopInline( TidyDocImpl* doc, Node *node ) {
142: if (lexer → istacksize > 0) {...
147:    while (istack→attributes){ ...
151:     FreeAttribute( doc, av ); }
parser.c:
128: Node* DiscardElement( TidyDocImpl* doc, Node *element ) {
132: if (element){...
136:   FreeNode( doc, element); }
140:}
309: Node *TrimEmptyElement( TidyDocImpl* doc, Node *element ) {
311:  if ( CanPrune(doc, element) ){...
316:   return DiscardElement(doc, element); }
```

Figure 2.7: Double free example.

slice 0 by 60% and dependence edges by 62%. Therefore, effective use of partial logging can greatly reduce slice sizes.

2.2.2 Localizing a Double Free Bug

Tidy-34132 contains a double-free memory bug which manifests itself when the input HTML file contains a malformed *font* element, e.g., of the form `<font color="red"<?font>`. The relevant code for this bug is presented in Figure 2.7. The program constructs a *node* structure for each element (e.g., *font*) in the HTML file. An element may contain multiple attributes corresponding to the *attributes* field of the *node* structure, which is a pointer to the *attribute* structure. The program pushes a deep copy of the *node* structure onto the stack when encountering an inline element (i.e., *font* in our test case) by calling *PushInline* (line 057). The deep copy is performed by duplicating the dynamically allocated structure pointed by each field in the *node* structure as well as fields of fields recursively. For example, the program duplicates the *node*'s *attributes* fields and fields of the *attributes* structures, as shown in line 092. However, the program makes a shallow copy of the *php* field in the *attribute* structure by mistake in line 033 because of the missing statement, as shown in line 039. All the copies of *node* structure pushed onto the stack by *PushInline* will be subsequently popped out in function *PopInline* (line 097), where all the allocated regions will be freed recursively. In some situations, due to the shallow copy, the *php* fields of some *node* structures will contain dangling pointers. If some element in the HTML file is empty and can be pruned out, the program removes the node from the markup tree and discards it by calling *TrimEmptyElement* (line 309) which eventually calls *DiscardElement* on line 316. Node deletion is just a reverse process of node deep copy, i.e., free all the dynamic allocated memory regions in the *node* structure in a recursive fashion, including the structures pointed by the *php* fields. With some special HTML files as input, the program crashes when it tries to trim the empty *font* element because the *php* field of the *attributes* field of the *font* element has been freed in *PopInline*.

Since the bug is very complicated, debugging is very time-consuming with traditional debuggers. Programmers can identify the bug much easier with the help of our debugger. Although a double-free bug may manifest itself far away from the second *free*, the program happens to crash at the second *free* in our test case. As mentioned before, a program crash at *free* can be caused by three kinds of bugs—double free, unmatched free, or heap buffer overflow. The programmer can use `breakpoint` *free/malloc/memcpy/strcpy* `if` *condition* to identify the exact bug type. In our test case, the command `breakpoint` *free* `if arg`*1*`==`*second_free_ptr* captures the position of the first free quickly. The zoom component of our approach now reveals its power, as the reported crash point can be far from the second *free*. The programmer uses the `slice` command to get the dynamic slice of the memory units used at the crash point and then pinpoint the root cause with the state alteration, inspection and rollback interfaces introduced in our debugger.

As we can see, fixing this bug (line 039 in istack.c) calls for far more program comprehension than the positions of the two *free* calls (line 136 in parser.c), which is the best bug report that existing automatic debugging tools (e.g., Memcheck [79]) can achieve. Suppose the programmer has already known the positions of the two *free* calls with the help of either our debugger or automatic debugging tools. To figure out under which condition the bug manifests itself and then remove the defect, the programmer still needs to resort to debuggers. This example illustrates a normal situation where automatic debugging techniques lag far behind the requirements raised from practical debugging. They can only be a supplement to debuggers rather than a substitute.

The programmer can quickly gain program understanding and fix bugs with the help of our debugger in this case. Suppose the programmer has known the position of the two *free*'s, with the help of either our debugger or other automatic debugging tools. First, the programmer can generate a dynamic slice for the two variables used in the first and second *free* respectively. Then she can easily find out where the shallow copy comes from by following the data dependence edges related to those variables. For example, by following only two hops along the data dependence edges in the generated slice for the variable used in the first *free*, she can find out that the shallow copy comes from line 033. However, the *DupAttrs* function is expected to generate a deep copy of a given attribute, then the programmer figures out that some statements which should generate the deep copy is missed in this function and she can fix this bug quickly.

The programmer can also leverage state alteration to gain more program comprehension and then fix the bug. For example, she can use the `switch` command to switch some predicates in the dynamic slice of the crash point (the second free) for better understanding the program behavior and crash condition. For this particular bug, switching the last execution instance of the predicate at line 132, 142, or 311 will make the program function properly. Hence, we can infer that a combination of those predicate is the bug manifestation condition and the bug will disappear with any predicate unsatisfied. The programmer can also suppress some functions in the slice of the crash point to isolate the bug. Suppression of the final invocation of *TrimEmptyElement*, *PopInline*, *PushInline* or *DiscardElement* in our test case can eliminate the crash, which suggests that an abnormal data flow is avoided by following any of the execution suppression. From the result of state alteration, the programmer will know that the program crashes if the last processed element is an inline and prunable element. This provides valuable hints to the programmer.

In the next chapter we show how this double free bug can be detected with trivial programming efforts (3 lines of bug specification), and located much faster with the help of value propagation chains which allow programmers to only examine 16 instructions (vs. 4,687 instructions with dynamic slicing).

2.2.3 Localizing a Heap Buffer Overflow Bug

Figure 2.8 shows a bug in *bc-1.06*. The program fails with a memory corruption error at line 557. The root cause is the incorrect predicate at line 176, where the variable *v_count* is misused instead of the desired one, *a_count*. The variable *v_count* stands for the total number of variables seen so far, while the variable *a_count* represents the size of the variable *arrays* which stores a structure for each array seen so far. When the

```
storage.c:
153: void more_arrays () {...
166: a_count += STORE_INCR;
167: arrays = bc_malloc (a_count*sizeof(bc_var_array *));
     /* correct one: for (; indx < a_count; indx++)*/
176: for (; indx < v_count; indx++)
177:   arrays[indx] = NULL;
util.c:
542: int lookup ( char *name, int namekind) {...
553: id = find_id (name_tree, name);
554: if (id == NULL){
     /* We need to make a new item. */
557:   id = (id_rec *) bc_malloc (sizeof (id_rec)); /*crash point*/
     ...}
566: switch (namekind){
569:   case ARRAY:
576:     id→a_name = next_array++;
577:     a_names[id→a_name] = name;
578:     if (id→a_name < MAX_STORE) {
580:       if (id→a_name >= a_count)
581:         more_arrays ();
```

Figure 2.8: Heap buffer overflow example.

program encounters a new array and the variable *arrays* is full, it will call the function *more_arrays* to dynamically reallocate a larger array (line 167), copy data from the old array to the new array, and initialize all unused units to $NULL$ (line 177). When there are more variables than the size of *arrays* (i.e., *v_count* > *a_count*), the heap object *arrays* is overflowed and the metadata maintained by the heap allocator is corrupted. After that, if the program encounters a new symbol, it allocates a new data structure for this symbol at line 557, where program crashes abnormally because of the corrupted metadata.

To figure out what causes the memory corruption, the programmer can restart the program, save checkpoints at some early points, and start tracing to enable the dynamic slicing. She can repeat the program execution until the program crashes again. She then can get the dynamic slice of the memory units used at the crash point (*0x0805dcc0* in our test case). Unexpectedly, the crash point directly depends on the statement at line 177, which is supposed to write $NULL$ to some units in *arrays*. The programmer cannot find any relationship between the statement and the crash point if she only follows the source code level control/data flow. The captured hidden dependence provides effective guidance to the programmer who then can expedite the debugging procedure by focusing on the abnormal dependence in the slice.

To verify whether the statement at line 177 is the root cause, the programmer can go back to a previous execution point by `rollback` *checkpoint_id* and rerun the code before the statement. She then can use the `suppress` command to suppress execution of the specific instance of statement 177 in the slice of the crash point. She may also switch the predicate at line 176 to suppress the execution of statement 177. As it turns out, the program works properly with statement 177 suppressed. Therefore, now the programmer knows that either statement 177 or the statements influencing it (e.g., line 176) are wrong. By concentrating on the suppressed statement 177, she can easily find that the suppressed statement tries to write memory unit *0x0805dcc0* which is out of the scope of *array* and thereby should not be modified here. Next, the programmer can focus on the statements which affect the execution of statement 177 using the `slice` command with statement 177 as slicing criterion. The root cause of the heap buffer overflow bug can be pinpointed easily by examining the first statement (statement 176) in the generated slice.

Next chapter illustrates how the heap buffer overflow bug in *bc-1.06* can be detected and located much faster via declarative specification of bug conditions and their root causes, and automatic generation of debugger code. Chapter 4 instead shows how this long failure inducing input (1310 chars) can be greatly simplified (190 chars) via relevant input analysis based delta debugging techniques, then instead of the original

```
ttobjs.c:
213: #define ALLOC_ARRAY(ptr, old_count, count, type) \
214:   (old_count >= count ? 0 : \
215:   !(free_aux(mem, ptr), ptr = \
216:   mem→alloc_bytes(mem, (count) * sizeof(type),"ttobjs.c")))

294: Context_Create( void* _context, void* _face) {
302: PExecution_Context exec =(PExecution_Context)_context;  ...
319: if( ALLOC_ARRAY(exec→glyphIns, exec→maxGlyphSize,
319: maxp→maxSizeOfInstructions, Byte) || ...)
357:   goto Fail_Memory ;
gsalloc.c
717: i_free_object(gs_memory_t * mem, void *ptr, ...) {
728: pp = (obj_header_t *) ptr - 1;
729: pstype = pp→o_type;     ...
770: finalize = pstype→finalize; /*crash point*/
igc.c:
157: gs_gc_reclaim(vm_spaces * pspaces, bool global)
ttinterp.c:
708: in Ins_ROLL () {
719: args[0] = B;
```

Figure 2.9: Dangling pointer dereference example.

long input (1310 chars), programmers start the debugging task with the simplified input (190 chars).

2.2.4 Localizing a Dangling Pointer Dereference

A dangling pointer dereference bug in *Ghostscript-8.12* is shown in Figure 2.9. In this buggy program, all execution contexts share the same glyph buffer (*exec→glyphIns* at line 319). When an execution completes, its context data structure will be reclaimed in the garbage collection function *gs_gc_reclaim* at line 157 in file *igc.c*. However, the shared glyph buffer and its metadata structure *obj_header_t* are reclaimed improperly. The o_type field of the *obj_header_t* struct is a union of an integer and a pointer to the *gs_memory_struct_type_s* structure. Before the context is reclaimed, the *o_type* field functions as a pointer to the *gs_memory_struct_type_s* structure storing a finalization function for the glyph buffer. After the context is reclaimed, it is reused (or destroyed) as an integer, and written in function *Ins_ROLL* at line 719.

When a new execution context is created in function *Context_Create* at line 294, the original glyph buffer is freed in function *free_aux* at line 215 if the current glyph buffer's size is less than the max size of instructions in the newly created context. The program calls *i_free_object* at line 717 and a larger glyph buffer is created at line 216. However, because the shared glyph buffer has been reclaimed when the previous context completed (at line 157), the program crashes when it tries to free the glyph buffer at line 717 by referencing a dangling pointer *pstype* at line 770. The pointer *pstype* is supposed to store a pointer to the *gs_memory_struct_type_s* structure that keeps a finalization function for the glyph buffer. When it is reused wrongly as an integer, it is first written at line 157 in file *igc.c* and next at line 719 in file *ttinterp.c*.

As we can see, this bug is very complicated (e.g., the depth of call stack at the crash point is as high as 20). Unfortunately, such complexity is typical of many real-world bugs, and few automatic debugging techniques can help find and fix them. As a consequence, programmers still have to use debuggers to pinpoint the root cause and rectify the bug. As we will illustrate shortly, our debugger greatly reduces the programmers' effort required to remedy this bug.

To debug the program, the programmer can restart the program execution and create a checkpoint at some early point before the crash point (e.g., line 770 in file *ttobjs.c*). Suppose the program crashes at the nth execution of function *i_free_object* (n=133,177 in our test case). The programmer may want to suppress the whole function execution using command `suppress` *i_free_object 133177* or switch the last encountered predicate (at line 214). The program will work in the right manner after the execution suppression or predicate switching, which implies that the metadata of the *glyph*

```
create.c:
800: create_archive (void){...
806: if (incremental_option){...
812:   collect_and_sort_names ();
names.c:
713: add_hierarchy_to_namelist (struct name *name, dev_t device){
715: char *path = name→name;
716: char *buffer = get_directory_contents (path, device);
765: ...}
772: collect_and_sort_names{...
809: add_hierarchy_to_namelist (name, statbuf.st_dev);
820: ...}
increment.c:
173: get_directory_contents (char *path, device) {...
     /* for scanning directory*/
180: char *dirp = savedir (path);    ...
204: if (children != NO_CHILDREN)
205:   for (entry = dirp;
206:     entrylen = strlen (entry))!= 0;//crash point
207:     entry +=entrylen +1)
         {...}
```

Figure 2.10: NULL pointer dereference example.

buffer may be modified unexpectedly somewhere and thereby cannot be used to free the *glyph* buffer. Next, the programmer can focus on why and where the metadata are unexpectedly modified.

To find out where the metadata are modified, the programmer can use the `slice` command to get the slice of the variable *pstype*. With the help of generated slice, he can easily figure out that *pstype* is data dependent on the variable *pp* and *pp→o_type* at line 729, and the variable *pp→o_type* depends on the write statement at line 719 in function *Ins_ROLL*, which is abnormal since the *pp→o_type* field should function as a pointer to the *gs_memory_struct_type_s* structure now and it should not be written in the *Ins_ROLL* function, where it serves as an integer. Based on this valuable clue, programmers can track the root cause at line 157 by iterating these steps.

2.2.5 Localizing a NULL Pointer Dereference

A NULL pointer dereference bug in *Tar-1.13.25* causes the program to crash when the user tries to do an incremental backup of a directory without having read access permission to it. The sketch of the bug is shown in Figure 2.10. When option *"-g"* is used to create an incremental backup, the program execution will follow the true branch at line 806. The function *collect_and_sort_names* then constructs a sorted directory tree by indirectly calling the function *get_directory_contents* to collect all the files recursively for each directory given in the command line. If the user does not have read access to the specified directories, the function *savedir* will unexpectedly return a NULL pointer, causing the program to crash at line 206 in function *strlen*. Generally, the source code/debugging info for system libraries (e.g., *strlen* here) is unavailable to programmers/debuggers, making it difficult for programmers to understand the bug by following/single-stepping the source code execution.

With the help of our debugger, the programmer can get significant insight into the cause of the failure in a very efficient manner. First, he can construct a dynamic slice of the crash point (e.g., line 206). Then he can find out where this NULL pointer comes from by following only two hops along the data dependence edges in the slice. Alternatively, he can leverage state alteration to understand the crash condition and then fix it. He may choose some suspicious predicates in the slice to switch. For instance, the program does not crash if the predicate at line 806 or line 204 is switched. The programmer will know that this bug manifests itself when both predicates evaluate to true, and it disappears when any of them does not hold. In fact, this bug does only concern incremental backup. Furthermore, the failure disappears by suppressing the fifth invocation of *add_hierarchy_to_namelist* or the second invocation of function *get_directory_contents*,

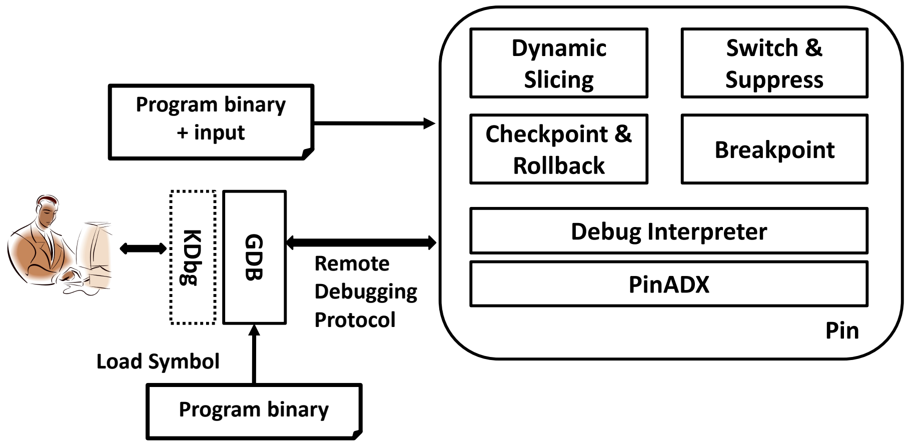


Figure 2.11: Components of the `Qzdb` debugger.

both of which are in the backward slice of the crash point. To better identify the failure, programmer can shrink the suppression scope by only suppressing some lines in the crash function. All the clues can help the programmer understand the crash condition and bug nature, and finally fix it. That is to say, dynamic slices help the programmer to find out unexpected or abnormal data dependence, while state alteration and state rollback interfaces help him to quickly understand the failure condition and bug nature, and finally fix it.

Next chapter shows how this bug can be detected with the help of 1 line of bug specification, and located efficiently (just examine 4 instructions) via declarative bug specification and debugger generation.

2.3 Implementation

The prototype implementation of the `Qzdb` interactive debugging strategy, shown in Figure 2.11, consists of GDB-based [26] and Pin-based [62] components. The user interacts with the GDB component via a command line interface or a KDbg [48] based graphical interface. The Pin-based component implements our new debugging commands. The GDB component communicates with the Pin-based component via PinADX [61], a debugging extension of Pin. The Pin-based component implements the new capabilities via dynamic binary instrumentation. The extended KDbg provides an intuitive interface for switching predicates, suppressing execution, setting breakpoints, turning recording on/off, and inspecting and stepping through slices.

Predicate switching. Upon receiving `switch` commands, we use Pin to first invalidate existing instrumentation involving specified code regions and then reinstrument the code to switch the results of predicates by swapping their fall-through and jump targets.

Execution suppression. After the programmer issues a `suppress` command, existing instrumentation is invalidated and new instrumentation is added to skip over the suppressed execution instance of the instruction. The instruction is executed normally if it is not the suppressed execution instance. If all instances are to be suppressed, the instruction is deleted using Pin.

Dynamic slicing. We implement the `slice` command by instrumenting the code to record the PC, dynamic instance, as well as memory region(s) and register(s) read and written by instructions. We instrument both user and library code. We turn off recording when `record off` is encountered. For limiting the time and space overhead of dynamic data dependence graph construction, we use the limited preprocessing (LP) method by Zhang et al. [120]. For accurately capturing dynamic control dependences, we use the online algorithm by Xin and Zhang [106]. The immediate postdominator information is extracted using Diablo [16].

Conditional breakpoint. To implement the extended conditional breakpoint command we invalidate the existing instrumentation and then reinstrument each function call to first check whether the function name is the same as the specified *lib_func*.

If so, the instrumentation code evaluates the given *condition* (if any) and triggers a generated breakpoint if it is satisfied.

Checkpointing and rollback. Undo-log based incremental checkpoints [50] are adopted to keep only the modifications between two checkpoints and save space. When a `checkpoint` command is received, we first save the state of all registers maintained by Pin. Subsequently we record the original value of each modified memory cell by instrumenting each memory write operation. Upon a `rollback` command, we restore the logged values to their memory cells and registers. Because Pin cannot track into system calls, we handle system calls and I/O as follows. The system calls' side effects are detected by analysing commonly-used system calls and recording the memory regions read/written by each system call. For file I/O, whenever a checkpoint is generated, we record the file pointer positions for all open file descriptors. When the program is rolled-back, we restore file pointer positions, so file reads and writes proceed from correct offsets on reexecution. We do not handle interactive I/O specially, but rather offer the expected semantics for reexecution. For example, for the console, after a roll-back, the user must type the input again, and output messages will be printed again. In our experience, this approach works well in practice.

2.4 Performance Evaluation

Next we show that the time and space overheads for state alteration, inspection, and rollback are acceptable for interactive debugging. To quantify these overheads, we have conducted experiments with the programs from Table 2.2. Since our objective is to measure time and space costs, we used a passing test case to run each application to completion. Table 2.3 shows the run characteristics including number of executed

Program	**Test Case Description**	**Dynamic Executed Instructions**	**Null Pin** Time (sec)
ncompress-4.2.4	compress a folder(148KB)	10278947	0.33
Tidy-34132	check a HTML file(104 lines)	2125726	0.56
bc-1.06	interpret a source file(121 lines)	1846427	0.44
Ghostscript-8.12	PS to PDF conversion(18KB)	3909749	0.47
Tar-1.13.25	create an archive(789K)	4654490	0.51

Table 2.3: Run characteristics.

Program	**Baseline**	**Pay-Once Time and Space Overhead**					**Slice Time Overhead**		
	Time	DU		CD		LP	(sec)		
	(sec)	Time (sec)	Space (MB)	Time (sec)	Space (MB)	Time (sec)	AVG	MIN	MAX
ncompress	0.33	5.83	93.63	2.88	63.19	3.28	37.19	13.56	71.52
Tidy	0.56	9.45	12.81	4.62	17.60	0.24	31.14	11.43	46.59
bc	0.44	5.58	11.52	2.63	13.51	0.20	14.44	11.53	21.70
Ghostscript	0.47	8.28	24.43	4.93	25.58	0.53	20.44	2.95	45.04
Tar	0.51	7.23	24.78	3.33	25.14	0.06	6.69	7.74	15.04

Table 2.4: Slicing time and space overhead.

Program	**Time Overhead** MS/(1K instructions)	**Space Overhead** KB/(1K instructions)
ncompress-4.2.4	0.85	15.62
Tidy-34132	6.62	14.65
bc-1.06	4.45	13.88
Ghostscript-8.12	3.38	13.10
Tar-1.13.25	2.27	10.98
Average	*3.51*	*13.65*

Table 2.5: Time and space overhead: DU & CD

instructions and the "Null Pin"[1] running time. All experiments were conducted on a DELL PowerEdge 1900 with 3.0GHz Intel Xeon processor and 3GB RAM, running Linux, kernel version 2.6.18.

2.4.1 Slicing Overhead

The time and space overhead for slicing are presented in Table 2.4. For each program, we turned on the recording to collect definition/use information and detect dynamic control dependences for the whole execution. We then applied limited prepro-

[1]Null pin running time means running time with Pin without any instrumentation.

cessing (LP) to the generated def-use information to get a summary of all downward exposed definitions of memory addresses and registers for each trace block. Using the generated trace and summary, we computed slices for the last twenty statements.

The *Pay-Once Time* and *Space Overhead* columns 3–7 in Table 2.4 show the time and space overhead which is only incurred once and amortized over all subsequent slice computations. The pay-once time overhead is further broken down into time overhead for recording definition/use (*DU*), control dependence (*CD*), and preprocessing of the generated def-use information (*LP*). The pay-once space overhead is broken down into space overhead for definition/use information recording (*DU*) and control dependence (*CD*) as the space overhead for LP is relatively insignificant.

The average (*AVG*), minimum (*MIN*), and maximum (*MAX*) slice computation times are given in the *Slice Time Overhead* column. We observe that the time overhead for slicing is not greatly dependent on the position of the slice criterion. Instead, it is dominated by the nature of the slice criterion and program behavior. Most slice computations can be done in 1 min., which is acceptable considering the large amount of time spent on debugging.

The time and space overhead of both def-use information recording and control dependence detection per 1K instructions are given in the second and third columns of Table 2.5, respectively. The time overhead ranges from 0.85ms to 6.62ms per 1K instructions and the average overhead is 3.51ms per 1K instructions. The space overhead ranges from 10.98KB to 15.62KB per 1K instructions and the average overhead is 13.65KB per 1K instructions. We believe that the pay-once time and space overheads for dynamic slicing are acceptable.

Program	**# Checkpoints**	**Time** (sec)	MS/1K instructions	**Space** (KB)	KB/1K instructions	**Rollback Time** (MS)
ncompress	11	8.93	0.87	28547.6	2.78	356.17
Tidy	3	9.31	4.38	233.4	0.11	4.02
bc	2	6.06	3.28	45.1	0.02	0.04
Ghostscript	4	8.52	2.38	788.9	0.20	12.30
Tar	5	7.40	1.80	189.6	0.04	0.22

Table 2.6: Checkpointing and rollback time and space overhead.

2.4.2 Checkpointing Overhead

The time and space overhead of checkpointing are given in Table 2.6. This data corresponds to checkpointing every one million instructions. The second column shows the number of checkpoints generated. The total program execution time with incremental checkpointing is given in the third column. The fifth column presents the total space overhead of the generated checkpoints. The time needed to rollback a program from the end to the beginning, which represents the largest distance the programmer can rollback the program, is shown in the last column. The benchmarks reveal that, the larger the size of the generated checkpoints, the longer it takes to rollback the program; *ncompress* incurs the largest space overhead for the 11 checkpoints and it requires the longest time to rollback the program to the beginning.

The time and space overhead of incremental checkpointing per 1K instructions is given in the fourth and sixth columns of Table 2.6, respectively. As we can see, the time overhead ranges from 0.87ms to 4.38ms per 1K instructions, while the space overhead ranges from 0.02KB to 2.78KB per 1K instructions. Compared to the time and space overhead of recording and control dependence shown in Table 2.5, the time and space overhead per 1K instructions for incremental checkpointing is much lower. This is because only memory write instructions need to be lightly instrumented for incremental checkpointing, while both memory and register read and write instructions need to be

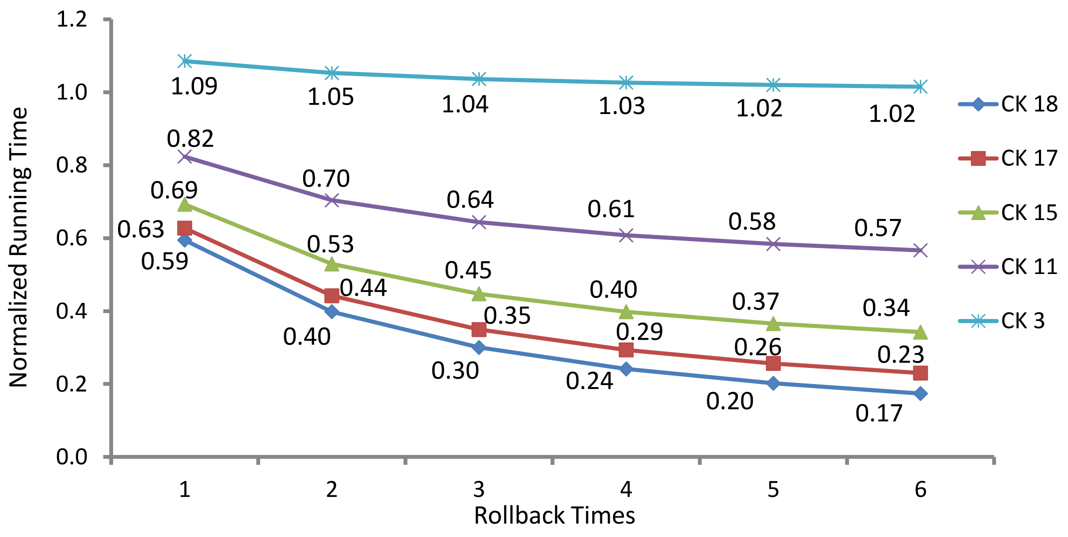


Figure 2.12: Runtime savings due to rollback.

heavily instrumented for both def-use information recording and control dependence detection.

2.4.3 Efficiency of State Rollback

As mentioned in Section 2.1.3, our state rollback command replaces the rolled-back part of execution by altered program execution (e.g., due to feeding it a different input or switching control flow) in the log. Thus, programmers have no need to rerun the program from the beginning. Of course, to rollback the program state, programmers have to incur the checkpointing overhead during the initial full run. In this experiment, we emulate a traditional debugging process and compare the running time with and without use of state rollback. We consider a run of *bc* that takes 118 seconds in Null Pin mode and executes 36.2×10^{10} instructions. A checkpoint is made every $2 * 10^{10}$ instructions leading to 19 checkpoints numbered from 0 to 18. We compare the execution time with use of rollback to different checkpoints (CK 3, CK 11, CK 15, CK 17, CK 18) for varying number of times (1 through 6) with the execution time without use of rollback. The execution times with rollback, normalized with respect to the corresponding times

Program	**Baseline** (seconds)	Suppress with Recording and Checkpointing (seconds)	Suppress with Checkpointing Only (milliseconds)
ncompress	0.33	0.60 (182.13%)	3.33 (1.01%)
Tidy	0.56	0.13 (22.51%)	24.00 (4.29%)
bc	0.44	0.23 (52.02%)	6.67 (1.52%)
Ghostscript	0.47	0.35 (74.04%)	24.80 (5.28%)
Tar	0.51	0.09 (18.21%)	1.85 (0.36%)

Table 2.7: Suppression time overhead.

without rollback, are shown in Figure 2.12. We observe that the execution time savings due to use of rollback are substantial and higher for more recent checkpoints (e.g., CK 18) and the savings increase with the number of times rollback is performed (e.g., rollback six times). However, if the rollback is performed to an early checkpoint (e.g., CK 3), its benefit disappears.

2.4.4 State Alteration Overhead

The time overhead of execution suppression is given in Table 2.7. We consider two scenarios. The first scenario simulates the case where the programmer suppresses a statement with both recording and incremental checkpointing turned on, while in the second scenario only incremental checkpointing is turned on. The data presented is averaged over suppressing 10 statements spread around the middle of the execution. We observe that performing execution suppression in the first scenario incurs substantially higher runtime overhead. This is because in this scenario an execution suppression command invalidates many existing instrumentations, leading to more future reinstrumentation costs, and higher runtime overhead. From Table 2.7, we can see that the average time overhead incurred by execution suppression ranges from 0.36% to 182.13% compared to the baseline. This overhead is acceptable and a worthy trade-off for the benefits of our approach. We omit presenting the overhead for predicate switching as it is similar to the overhead of execution suppression due to similarities in their implementations.

2.5 Summary

This chapter has presented the `Qzdb` debugger for debugging singlethreaded programs; `Qzdb` offers powerful state alteration and state inspection capabilities. State alteration commands enable programmers to narrow down the potential faulty code and ascertain their conjectures efficiently. State inspection commands enable programmers to comprehend program behavior and the nature of the bug rapidly. In addition to the above high-level commands, `Qzdb` also supports all low-level commands supported by `GDB`. Case studies on real reported bugs as well as performance evaluation demonstrate the effectiveness and efficiency of `Qzdb`.

The next two chapters are aimed at making `Qzdb` extensible and efficient respectively. To make the basic debugging framework extensible (especially for memory-related bugs) a new approach is presented for constructing debuggers based on declarative specification of bug conditions and root causes, and automatic generation of debugger code. To make the debugging framework more efficient, a new dynamic analysis called relevant input analysis is developed for enabling input and execution simplification.

Chapter 3

Integrating Specialized Debuggers in `Qzdb` via Bug Specifications

This chapter presents a novel approach that allows `Qzdb` to be extended via integration of algorithms for detection of specific kinds of bugs. This approach is illustrated by considering memory-related bugs. Since the detection of memory-related bugs is tedious using general-purpose debuggers [26, 51, 113, 31], programmers use tools tailored to specific kinds of bugs (e.g., buffer overflows [71, 18], dangling pointer dereferences [19], and memory leaks [108, 83]); however, to use the appropriate tool the programmer needs to first know what kind of bug is present in the program. Second, when faulty code is encountered during execution, its impact on program execution might be observed much later (e.g., due to a program crash or incorrect output), making it hard to locate the faulty code. Third, debuggers are also written by humans, which has two main disadvantages: (a) adding support for new kinds of bugs entails a significant development effort, and (b) lack of formal verification in debugger construction makes debuggers themselves prone to bugs.

The above challenges are addressed in this chapter using a novel approach for constructing debuggers for memory-related bugs. We allow bugs and their root causes to be specified declaratively, using just 1 to 4 predicates, and then use automated translation to generate an actual debugger that works for arbitrary C programs running on the x86 platform. We have proved that bug detection is sound with respect to a low-level operational semantics, i.e., bug detectors fire prior to the machine entering an error state. Our work introduces several novel concepts and techniques, described next.

Declarative debugger specification. In our approach, bugs are specified via *detection rules*, i.e., error conditions that indicate the presence of a fault, defined as First-order logic predicates on abstract states. In Section 3.1 we show how bug specifications can be easily written. Using detection rules as input, we employ automated translation to generate the debugger implementation; thanks to this translation process, explained in Section 3.3.1, from 8 lines of specification about 3,300 lines of C code are generated automatically.

Debugger soundness. We use a core imperative calculus that models the C language with just a few syntactic forms (Section 3.2.1) to help with specification and establishing correctness. We define an operational semantics (Section 3.2.2) which models program execution as transitions between abstract states Σ; abstract states form the basis for specifying debuggers in a very concise yet effective way. Next, we define error states for several memory bugs, and use the operational semantics (which contains transitions to legal or error states) to prove that the detectors are sound (Appendix A).

Value propagation chains. In addition to bug detection rules, our specifications also contain *locator rules*, which define value propagation chains pointing to the root cause of the bug. These chains drastically simplify the process of detecting and locating the root cause of memory bugs: for the 8 real-world bugs we have applied our approach to, users only have to examine just *1 to 16* instructions to locate the bugs (Section 3.4.2).

Prior efforts in this area include memory bug detectors, algorithmic debugging, and monitoring-oriented programming; we provide a comparison with related work in Chapter 6. However, to the best of our knowledge, our work is the first to combine a concise, declarative debugger specification style with automatic generation of bug detectors and locators, while providing a correctness proof.

The approach presented has the following advantages:

1. *Generality.* As we show in Section 3.1, bug specifications consist of 1 to 4 predicates per bug. Thus, specifications are easy to understand, scrutinize, and extend. Formal definitions of program semantics and error states show that bug detection based on these bug specifications is correct.
2. *Flexibility.* Instead of using specialized tools for different kinds of bugs, the user generates a single debugger that still distinguishes among many different kinds of bugs. Moreover, bug detectors can be switched on and off as the program runs.
3. *Effectiveness.* Bug detectors continuously evaluate error conditions and the user is informed of the error condition (type of bug) encountered before it manifests, e.g., via program crash. Bug locators then spring into action, to indicate the value chains in the execution history that are the root causes of the bug, which allow bugs to be found by examining just a handful of instructions (1 to 16), a small fraction of the instructions that would have to be examined when using dynamic slicing.

3.1 Bug Specification

Figure 3.1 provides an overview of our approach. As the program executes, its execution is continuously monitored and x86 instructions are mapped to low-level operational semantics states Σ (described in Section 3.2.2). For most memory bugs, programmers use an abstraction of the semantics (execution trace σ and redex e), to write bug specifications; the full semantics is available to specify more complicated bugs.

Bug detectors and bug locators are generated automatically from specifications. During debugging, detectors examine the current state to determine when an error condition is about to become true, i.e., the abstract machine is about to enter an error state. When that is the case, locators associated with that error condition report the error and its root cause (location) to the programmer. Our debugger is able to simultaneously detect multiple kinds of bugs, as illustrated by the stacked detectors and locators in the figure.

We now present the user's perspective to our approach. Specification is the only stage where the user needs to be creatively involved, as the rest of the process is automatic, thanks to code generation. We first describe the specification process (Section 3.1.1). Next, we illustrate how our approach is used in practice for memory bugs (Section 3.1.2) and other kinds of bugs (Section 3.1.2.4). Later on (Section 3.4.2), we demonstrate the effectiveness of our approach by comparing it with traditional debugging and slicing techniques.

3.1.1 Specifying Debuggers via Rules

Traces and redexes. To simplify specification, for most memory bugs, the programmers can describe bugs by just referring to traces σ and redexes e. The trace σ records

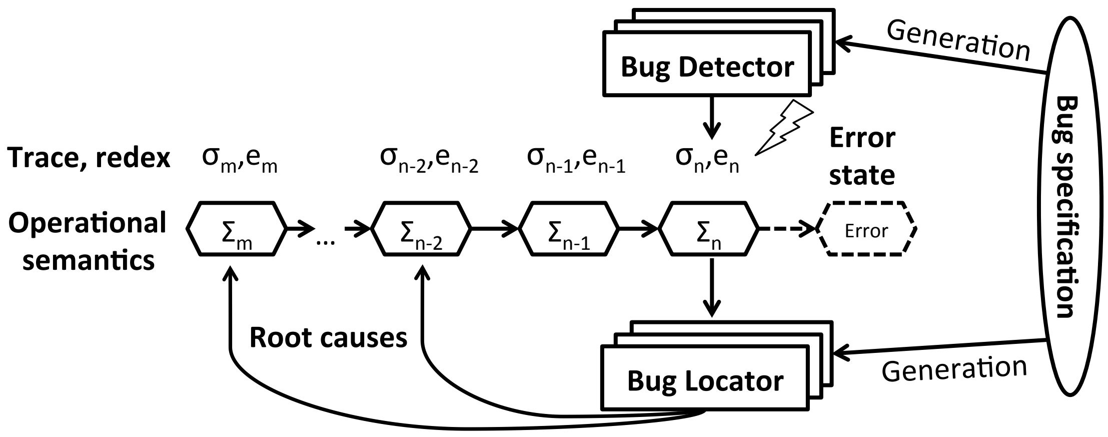


Figure 3.1: Overview of bug specification, detection and location.

the execution of relevant memory operation events—write for memory writes, malloc for allocation, free for deallocation—which are germane to memory bugs. Redexes e indicate the expression to be reduced next, such as function entry/exit, allocation/deallocation, memory reads and writes; when e is a memory operation, it contains a location r signifying the pointer to be operated on, e.g., freed, read from, or written to. The scarcity of syntactic forms for redexes and execution trace events provide a simple yet powerful framework for specifying C memory bugs.

Rules. To specify a bug kind, the user writes a rule (triple):

$$<\textit{detection point},\ \textit{bug condition},\ \textit{value propagation}>.$$

The first two components, *detection point* and *bug condition*, specify a bug detector, while the third component, *value propagation*, specifies a bug locator. Figure 3.2 shows how detection points, bug conditions and bug locators are put together to form rules and specify six actual classes of memory bugs. We now proceed to defining each component of a rule.

Detection Point	**Next Reduction e**	**Semantics**
$deref\_r\ r$	$*r$	memory read
$deref\_w\ r$	$r := v$	memory write
$deref\ r$	$*r/r := v$	memory access
$free\ r$	$\mathsf{free}\ r$	deallocation
$call\ z\ v$	$\mathsf{z}\ v$	function call
$ret\ z\ v$	$\mathsf{ret\ z}\ e$	function return

Table 3.1: Detection points.

Detection points specify the reductions where bug detection should be performed, as shown in Table 3.1. The programmer only needs to specify the detection point (left column). Our debugger will then evaluate the bug condition when the operational semantics's next reduction is e (middle column). For example, if the programmer wants to write a detector that fires whenever memory is read, she can use $deref\_r\ r$ as a detection point. Detection points which can match multiple reduction rules, coupled with the simple syntax of our calculus, make for brief yet effective specification; for example, using a single detection point, $deref\ r$, the user will at once capture the myriad ways pointers can be dereferenced in C.

Bug conditions are First-order logic predicates which allow memory bugs to be specified in a concise, declarative manner, by referring to the detection point and the trace σ. First, in Figure 3.2 (bottom) we define some auxiliary predicates that allow more concise definitions for bug detectors. $Allocated(r)$ checks whether pointer r has been allocated. The low-level semantics contains mappings of the form $r \mapsto (bid, i)$, i.e., from pointer r to the block bid and index i it points to; $Bid(r)$ returns r's block in this mapping. Therefore, $Allocated(r)$ is true if the block r is currently pointing into a block bid that according to the trace σ has previously been allocated, i.e., it contains a malloc event for this bid; '_' is the standard wildcard pattern. $Freed(r, r_1)$ is true if the block bid that r is currently pointing into has been freed, i.e., the trace σ contains a free event

Detection Rules	**Detection Point**	**Bug Condition**	**VPC**
[UNMATCHED-FREE]	$detect\langle\sigma; free\ r\rangle$:	$\neg Allocated(r) \vee r \neq Begin(r)$	$VPC(r)$
[DOUBLE-FREE]	$detect\langle\sigma; free\ r\rangle$:	$Allocated(r) \wedge Freed(r, r_1)$	$VPC(r)$, $VPC(r_1)$
[DANGLING-POINTER-DEREF]	$detect\langle\sigma; deref\ r\rangle$:	$Allocated(r) \wedge Freed(r, r_1)$	$VPC(r)$, $VPC(r_1)$
[NULL-POINTER-DEREF]	$detect\langle\sigma; deref\ r\rangle$:	$r = NULL$	$VPC(r)$
[HEAP-BUFFER-OVERFLOW]	$detect\langle\sigma; deref\ r\rangle$:	$Allocated(r) \wedge \neg Freed(r, \_)$ $\wedge(r < Begin(r) \vee r \geq End(r))$	$VPC(r)$
[UNINITIALIZED-READ]	$detect\langle\sigma; deref\_r\ r\rangle$:	$\neg FindLast(\_, \mathsf{write}, r, \_,)$	

Auxiliary predicates

$$\begin{array}{lcl} Allocated(r) & \doteq & \exists\ (\_, \mathsf{malloc}, \_, bid) \in \sigma : bid = Bid(r) \\ Freed(r, r_1) & \doteq & \exists\ (\_, \mathsf{free}, r_1, bid) \in \sigma : bid = Bid(r) \end{array}$$

Figure 3.2: Bug detection rules and auxiliary predicates.

for this *bid*. Note that free's argument r_1, the pointer used to free the memory block, is not necessarily equal to r, as r could be pointing in the middle of the block while r_1 is the base of the block (cf. Section 3.2.2).

With the auxiliary predicates at hand, we define First-order logic conditions on the abstract domain, as illustrated in the bug condition part of Figure 3.2. Note that $FindLast(ts, event)$ is a built-in function that traverses the trace backwards and finds the last matching event according to given signature. For example, a dangling pointer dereference bug occurs when we attempt to dereference r whose block has been freed before; this specification appears formally in rule [DANGLING-POINTER-DEREF], i.e., the bug is detected when the redex is $*r$ or $r := v$ and the predicate $Allocated(r) \wedge Freed(r, r_1)$ is true. Note that r_1 is a free variable here and its value is bound to the pointer which is used to free this block for the first time.

Bug locators. The last component of each rule specifies *value propagation chains* (VPC) which help construct bug locators. The VPC of variable v in a program state Σ is the transitive closure of value propagation edges ending at Σ for variable v. The

VPC is computed by backward traversal of value propagation edges ending at Σ for variable v. Note that dynamic slicing does not distinguish data dependences introduced by computing values from dependences introduced by propagating existing values. Value propagation edges capture the latter—a small subset of dynamic slices.

For each bug kind, the VPC specifies how the value involved in the bug manifestation relates to the bug's root cause. For example, in [DOUBLE-FREE], the root cause of the bug can be found by tracing the propagation of r (the pointer we are trying to free) and r_1 (the pointer that performed the first free). In [NULL-POINTER-DEREF], it suffices to follow the propagation of the current pointer r which at some point became $NULL$.

3.1.2 Memory Debuggers in Practice

We now provide a comprehensive account of how our approach helps specify, detect and locate the root causes of memory bugs using three examples of actual bugs in real-world programs—double free bug in *Tidy-34132* (previously studied in Chapter 2), and NULL pointer dereference bug in *Tar-1.13.25* (previously studied in Chapter 2), and unmatched free bug in *Cpython-870c0ef7e8a2*.

3.1.2.1 Double-free

Attempting to free an already-freed pointer is a very common bug. In Figure 3.2, the rule [DOUBLE-FREE] contains the specification for the bug: when the redex is `free` r and the predicates $Allocated(r)$ and $Freed(r, r_1)$ are both true, we conclude that r has already been freed.

In the previous chapter, we already shown how different types of commands introduced by `Qzdb` were used to detect a double free bug in *Tidy-34132*. Here we show

C code	Relevant events added to the trace σ

istack.c:
025: AttVal *DupAttrs(TidyDocImpl* doc, AttVal *attrs) {
032: newattrs = NewAttribute();
033: ***newattrs = *attrs;**
034: newattrs → next = DupAttrs(doc, attrs → next); ...
/*the following statement is missing in buggy code*/
Bug detection rules and auxiliary predicates
n 039**newattrs → php=attrs → php? \**
CloneNode(doc, attrs → php):NULL;
041:}
057: void PushInline(TidyDocImpl* doc, Node *node) {...
092: istack → attributes =
DupAttrs(doc, node → attributes);
094:}
097: void PopInline(TidyDocImpl* doc, Node *node) {...
147: while (istack→attributes){ ...
151: FreeAttribute(doc, av); }
parser.c:
128: Node*
DiscardElement(TidyDocImpl* doc, Node *element){
132: if (element){...
136: FreeNode(doc, element); }
140:}
309: Node *
TrimEmptyElement(TidyDocImpl* doc,Node *element){
311: if(CanPrune(doc, element)){...
316: return DiscardElement(doc, element);}
	malloc, n, 1_H
	write, p, $\_$,
	$return\ value\ of$ malloc
	write, $node$, $\_$, p
	write, $node$, $\_$, $node$
	write, php, $\_$, $node$
	write, $*php$, $\_$, php
	write, $attrs \rightarrow php$, $\_$, $*php$
	write, $newattrs \rightarrow php$, $\_$,
	$attrs \rightarrow php$
	write, $node$, $\_$, $newattrs \rightarrow php$
	write, mem, $\_$, $node$
	write, ptr, $\_$, mem
	free, ptr, 1_H
	write, $node$, $\_$, $attrs \rightarrow php$
	write, mem, $\_$, $node$
	write, ptr, $\_$, mem
	
	bug detected at free(ptr)

Value propagation chain

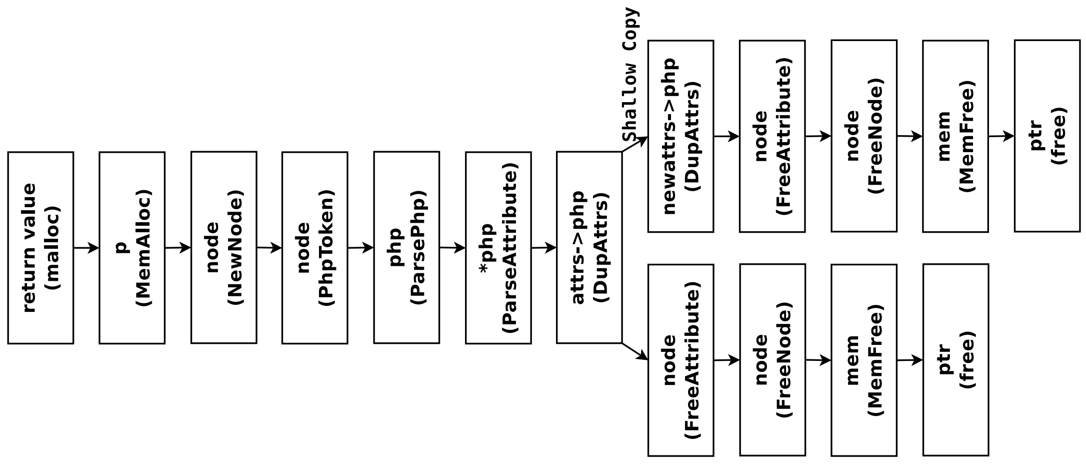


Figure 3.3: Detecting, and locating the root cause of, a double-free bug in *Tidy-34132*.

how this double free bug can be detected and located much faster with the techniques presented in this chapter. The relevant source code for this bug is presented in the left column of Figure 3.3. The program constructs a *node* structure for each element (e.g., *font*) in the HTML file. An element may contain multiple attributes corresponding to the *attributes* field of the *node* structure, which is a pointer to the *attribute* structure.

The program pushes a deep copy of the *node* structure onto the stack when encountering an inline element (i.e., *font* in our test case) by calling *PushInline* (line 057). The deep copy is created by duplicating the dynamically allocated structure pointed to by each field in the *node* structure as well as fields of fields recursively. However, the programmer makes a shallow copy of the *php* field in the *attribute* structure by mistake in line 033 because of a missing statement, as shown in line 039. All the copies of *node* structure pushed onto the stack by *PushInline* will be subsequently popped out in function *PopInline* (line 097), where all the allocated regions will be freed recursively. In some situations, due to the shallow copy, the *php* field of some *node* structures will contain dangling pointers. If some element in the HTML file is empty and can be pruned out, the program removes the node from the markup tree and discards it by calling *TrimEmptyElement* (line 309), which eventually calls *DiscardElement* at line 316. Node deletion is just a reverse process of node deep copy—it will free all the dynamically-allocated memory regions in the *node* structure in a recursive fashion, including the structures pointed to by the *php* fields. When providing certain HTML files as input, the program crashes when it tries to trim the empty *font* element because the *php* field of the *attributes* field of the *font* element has been freed in *PopInline.*

The second column of Figure 3.3 shows the events added to our trace σ during execution (irrelevant events are omitted). As we can see, the bug condition specified in rule [DOUBLE-FREE] is satisfied because σ contains events $\mathsf{malloc}, n, \mathit{1}_H$, and $\mathsf{free}, ptr, \mathit{1}_H$

(*1*$_H$ is the heap block id), indicating that block *1*$_H$ has been allocated and then freed, which makes $Allocated(r) \wedge Freed(r, r_1)$ true.

As presented in the previous chapter, the root cause of the double-free bug is the shallow copy in line 033, and the fix (line 039 in istack.c) calls for far more program comprehension (why, when and how the two different pointers wrongly point to the same heap block) than just the positions of the two *free* calls (line 136 in parser.c), which is the best bug report that current automatic debugging tools (e.g., Valgrind) can achieve. With the help of our bug locators, programmers need to examine just 16 instructions to figure out how and when the two pointers used in *free* point to the same memory region by following the value propagation chains for the two pointers (the two pointers can be the same in some situations, in which case the two value propagation chains are exactly the same). We show the value propagation chains for this execution in the bottom of Figure 3.3; in our actual implementation, this value chain is presented to the user. Note that the value of the pointer *ptr* used in the *free* function is first generated in function *malloc* and propagates to pointer *p* in function *MemAlloc*, and so on. The right child of node $\boxed{attrs \rightarrow php}$ is exactly the place where the shallow copy comes from (shallow copy from $attrs \rightarrow php$ to $newattrs \rightarrow php$). Hence, with the help of our bug locators, programmers can quickly understand the root cause and fix the bug (allowing programmers to only examine 16 instructions vs. 4,687 instructions with dynamic slicing, as shown in Section 3.4).

3.1.2.2 NULL Pointer Dereference

In Figure 3.2, the rule [NULL-POINTER-DEREF] is used to express and check for NULL pointer dereference bugs. We use the NULL pointer dereference bug in *Tar-1.13.25* (detailed in Chapter 2) to show how the simple rule [NULL-POINTER-DEREF]

C code	Relevant events added to the trace σ

savedir.c:
76: char * savedir (const char *dir){
DIR *dirp;
85: dirp = opendir (dir);
86: if (dirp == NULL)
87: return NULL;
129:..}
increment.c:
173: get_directory_contents (char *path){...
180: char *dirp = savedir (path); ...
205: for (entry = dirp; entrylen =
206: strlen (entry))!= 0; //**crash**
207: entry +=entrylen +1)
	
	write, $dirp$, _, savedir $retval$
	write, $entry$, _, $dirp$
	write, str, _, $entry$
	
	bug detected at strlen(str)

Value propagation chain

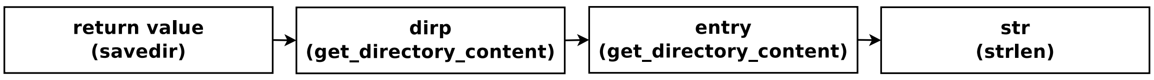


Figure 3.4: Detecting, and locating the root cause of, a NULL pointer dereference bug in *Tar-1.13.25*.

detects and locates the NULL pointer dereference bug. *Tar-1.13.25* crashes when the user tries to do an incremental backup of a directory without having read access permissions to it. A source code excerpt containing the bug is shown in the first column of Figure 3.4. If the user does not have read access to the specified directories, the function *savedir* will return a NULL pointer. This causes the program to crash at line 206 when passing this NULL pointer (*entry*) to function *strlen*.

With the help of our debugger, programmers can figure out the bug type, and get significant insight about the failure via bug locators. The trace of an execution which triggers this bug is shown on the right side of Figure 3.4. The NULL pointer bug detector will detect this bug when the NULL pointer is dereferenced in *strlen*. The value propagation chain of the NULL pointer, shown on the bottom of Figure 3.4, indicates where the NULL pointer originates (*line 87 in savedir.c*) and how it propagates to the crash point. Programmers can locate and fix this NULL pointer dereference bug very quickly with the value propagation chain. As we can see, compared to the debugging process presented in Chapter 2, the debugging process with automatically generated NULL pointer dereference bug detector and locator is much more targeted, thus much more effective.

3.1.2.3 Unmatched Free

Attempting to free an illegal pointer is a very common bug. In Figure 3.2, the rule [UNMATCHED-FREE] contains the declarative specification for the bug: whenever the evaluation reaches a point where the next expression is free r, if at least one of two conditions is met, the rule fires. If $Allocated(r)$ is false, the program tries to free something that has not been allocated in heap (e.g., blocks allocated in stack). If $r \neq Begin(r)$, the program attempts to free a pointer that has been allocated in heap, but instead of pointing to the malloc'd block (i.e., the base), r points somewhere in the middle of the block.

The real-world Python interpreter *Cpython-870c0ef7e8a2*, contains an unmatched free bug (freeing something that has not been allocated) that leads to a crash. The bug manifests when the *type.__getattribute__* function is misused (e.g., *type.__getattribute__(str, int)*) in the input Python program. The *type.__getattribute__(typeName, attrName)* function finds the attribute associated with *attrName* in *typeName*'s attribute list. However, passing a type name, e.g., *int*, as attribute name crashes the program.

A source code excerpt containing the bug is shown in the first column of Figure 3.5. Encountering a *type.__getattribute__(typeName, attrName)* statement, the Python interpreter invokes the *type_getattro* function at line 2483 to find the attribute associated with *name* in *type*'s attribute list at line 2517. When no attribute is found, an error message will be printed at line 2551 by calling *PyErr_Format*; *PyErr_Format* will eventually call *_PyUnicode_Ready* to prepare an Unicode string and print it. *_PyUnicode_Ready*

C code	Relevant events added to trace σ
unicodeobject.c:	
1353:_PyUnicode_Ready(PyObject *unicode){...	
1389: _PyUnicode_CONVERT_BYTES(...)	
1405: free((PyASCIIObject*)unicode→wstr);	
1479: ...}	
typeobject.c:	write, ptr, _,
2483: type_getattro(type, PyObject* name){	$unicode{\rightarrow}wstr$
/*the following statements are missing in buggy code*/	
2488: **if (!PyUnicode_Check(name)) {...**	**bug detected**
2492: **return NULL;}**	**at** free(ptr)
2517: attribute = _PyType_Lookup(type, name);	
2551: PyErr_Format(PyExc_AttributeError,	
2552: "type object '%.50s' has no attribute '%U'",	
2553: type→tp_name, name);	

Value propagation chain

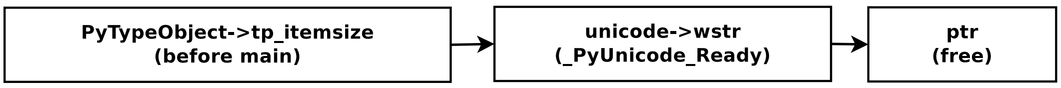


Figure 3.5: Detecting, and locating the root cause of, an unmatched free bug in *Cpython-870c0ef7e8a2*.

converts the Unicode string stored in *unicode→wstr buffer*, and then finally frees the buffer. However, the programmer has wrongly assumed that the *name* object at line 2483 must be an object of type *PyUnicodeObject* or subclass of it (e.g., *PyASCIIObject*), and has forgotten to add a type check at line 2488. When a type name is passed as the attribute name, the *unicode* at line 1405 is an object of type *PyTypeObject*, rather than *PyASCIIObject*. Thus, the programmer thinks *free* is invoked on *PyASCIIObject*'s *wstr* field when in fact it is invoked on *PyTypeObject*'s *tp_itemsize* field.

The second column shows the relevant events added to σ. As we can see there is no event malloc, n, _ to make $Allocated(r)$ true. The value propagation chain of ptr, shows how the wrong value of ptr is propagated from $unicode \rightarrow wstr$, which is a global variable and initialized before the execution of main, rather than dynamically allocated.

Rules	**Detection Point**	**Bug Condition**
[POSSIBLE-LEAK]	$detect\langle H; \sigma; \mathsf{ret}\ main\ v\rangle$:	$dom(H) \neq \emptyset$
[DEFINITE-LEAK]	$detect\langle H; P; \sigma; \mathsf{ret}\ main\ v\rangle$:	$\exists\ bid\ \in\ H : \neg(\exists\ r \mapsto (bid, \_)\ \in\ P)$
[LEAK-IN-TS]	$detect\langle H; \sigma; \mathsf{ret}\ ts\ v\rangle$:	$\exists\ bid\ \in\ H : FindLast(k, \mathsf{call}, ts, \_)$ $\wedge\ Time(bid) > k$
[GC-BUG]	$detect\langle H; F; \sigma; \mathsf{ret}\ gc\ v\rangle$:	$\neg(\ (\forall\ bid\ \in\ H : IsAlive(bid))$ $\wedge\ (\forall\ bid\ \in\ F : \neg IsAlive(bid))\ )$

Auxiliary predicates $\qquad IsAlive(bid) \quad \doteq \quad bid\ \&\ 0x1 = 1$

Figure 3.6: Bug detection rules and auxiliary predicates for other classes of bugs.

3.1.2.4 Other Classes of Bugs

While the core of our work is centered around the six classes of memory bugs we have just presented, programmers can use our approach to easily specify debuggers for other classes of bugs. We now proceed to briefly discuss examples of such classes; the bug specifications are presented in Figure 3.6.

Memory leaks. The rule [POSSIBLE-LEAK] specifies possible leaks as follows: if main is about to exit while the heap H contains one or more blocks that have not been freed, i.e., the heap domain is not empty, the rule fires.

With rule [DEFINITE-LEAK], we report leakages if, at the end of program execution, the heap H contains some blocks that no pointer in P points to. In other words, if there is no live pointer pointing to a block, we report the block as a definite leak.

The rule [LEAK-IN-TS] can be used to detect leaks in transactions. For simplicity, we assume that the scope of a transaction spans the entire body of a function denoted by metavariable ts. The programmer can easily specify that all the blocks allocated inside the transaction (body of ts) should be freed at the end of the transaction. We report leaks if, when function ts returns, the heap H contains some blocks which are allocated inside this function and have not been freed yet. Note that $FindLast(k, \mathsf{call}, ts, \_)$

matches the latest event which calls function ts, and the free variable k is bound to the timestamp for this event. $Time(bid) > k$ checks whether this block is allocated inside this function (or transaction).

Garbage collector bugs. [GC-BUG] illustrates how to specify one of the basic correctness properties for garbage collector implementations, that the alive bits are set correctly. Consider, for example, a mark-and-sweep garbage collector that uses the least significant bit of each allocated block to mark the block as alive/reachable (bit = 1) or not-alive (bit = 0). We can check whether the alive bits are set correctly at the end of a GC cycle before resetting them (bit = 0), as shown in rule [GC-BUG]: all blocks in H are marked as alive and all blocks in F are marked as freed.

3.2 Formalism

We now present our formalism: a core imperative calculus that models the execution and memory operations of C programs. We introduce this calculus for two reasons: (1) it drastically simplifies programmer's task of expressing bugs in C programs, by reducing the language to a few syntactic constructs and the dynamic semantics to a handful of abstract state transitions, and (2) it helps prove soundness.[1]

3.2.1 Syntax

We adopt a syntax that is minimalist, yet expressive enough to capture a wide variety of bugs, and powerful enough to model the actual execution. The syntax is shown in Figure 3.7. A program consists of a list of top-level definitions d. Definitions

[1]Soundness refers to detectors being correct with respect to the operational semantics to help catch specification errors; it does not imply that we certify the correctness of auto-generated and manually-written code for the Pin-based implementation, which operates on the entire x86 instruction set.

Definitions	d	$::=$	$\mathsf{main}\ e$
		$\mid$	$\mathsf{var}\ \mathsf{g} = v\ \mathsf{in}\ d$
		$\mid$	$\mathsf{fun}\ \mathsf{f}(x) = e\ \mathsf{in}\ d$
Expressions	e	$::=$	$v \mid x \mid \mathsf{let}\ x = v\ \mathsf{in}\ e$
		$\mid$	$\mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ e$
		$\mid$	$e; e \mid e\ e \mid \mathsf{ret}\ \mathsf{z}\ e$
		$\mid$	$\mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$
		$\mid$	$\mathsf{malloc}\ n \mid \mathsf{free}\ r$
		$\mid$	$*\, e \mid e := e \mid e +_p e \mid e + e$
Values	v	$::=$	$n \mid \mathsf{z} \mid r$
Global symbols	$\mathsf{f}, \mathsf{g}, \mathsf{z}$	$\in$	$GSym$
Indexes	i, j	$::=$	n
Pointers	r	$\in$	Loc
Integers	n		
Variables	x		

Figure 3.7: Syntax.

can be main, whose body is e, global variables g initialized with value v, and functions f with argument x (which is a tuple in the case of multiple-argument functions) and body e.

Expressions e can take several syntactic forms: values v, explained shortly; variable names x (which represent local variables or function arguments, but not global variables); let bindings; stack allocations $\mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ e$, where variable x is either a local variable or a function argument, n is its size (derived from the x's storage size), and e is an expression; sequencing $e; e$ and function application $e\ e$; function return $\mathsf{ret}\ \mathsf{z}\ e$; conditionals $\mathsf{if0}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$; $\mathsf{malloc}\ n$, allocating n bytes in the heap; $\mathsf{free}\ r$, deallocating a heap block; pointer dereference $*\, e$; assignment $e := e$; pointer arithmetic $e +_p e$, and integer arithmetic $e + e$. Values v can be integers n, global symbols z, or pointers r. Indexes, e.g., i, j, are integers and are used to specify the offset of a pointer in a memory block. Pointers r range over locations Loc, and are used as keys in a pointer map, as described next; note that we do not assume a specific type (e.g., integer, long) for pointers, as it is not relevant for defining the abstract machine.

Definitions

Block id	bid	$\in$	Bid
Block contents	b	$::=$	$\boxed{v_0, \ldots, v_{n-1}}$
Heap	H	$::=$	$\emptyset$
			$\mid bid \mapsto (b, n, k), H$
Freed blocks	F	$::=$	$\emptyset \mid (bid, h), F$
			$\mid (bid, s), F$
Stack frame	S	$::=$	$\emptyset$
			$\mid bid \mapsto (b, n, k), S$
Stack	$\overline{S}$	$::=$	$\emptyset \mid \overline{S}, S$
Pointers	P	$::=$	$\emptyset \mid r \mapsto (bid, i), P$
Timestamp	k	$::=$	n
Events	ev	$::=$	write, r, v, f
			$\mid n == n'$
			$\mid \mathsf{malloc}, n, bid$
			$\mid \mathsf{free}, r, bid$
			$\mid \mathsf{call}, z, v$
			$\mid \mathsf{ret}, z, v$
Timed events	ν	$::=$	(k, ev)
Traces	σ	$::=$	$\emptyset \mid \nu \cup \sigma$
Value origin	f	$::=$	$gen \mid z \mid r$
Expressions	e	$::=$	...

Evaluation contexts

$$\begin{array}{lll}
\mathbb{E} & ::= & [\,] \mid \mathsf{let}\ x = \mathbb{E}\ \mathsf{in}\ e \\
& \mid & \mathbb{E}\ e \mid v\ \mathbb{E} \mid \mathsf{ret\ z}\ \mathbb{E} \\
& \mid & \mathbb{E};\ e \mid v;\ \mathbb{E} \\
& \mid & \mathsf{malloc}\ \mathbb{E} \mid \mathsf{salloc}\ n\ \mathbb{E} \\
& \mid & \mathsf{free}\ \mathbb{E} \\
& \mid & \mathbb{E} := e \mid r := \mathbb{E} \mid *\mathbb{E} \\
& \mid & \mathbb{E} +_p e \mid r +_p \mathbb{E} \\
& \mid & \mathbb{E} + e \mid n + \mathbb{E} \\
& \mid & \mathsf{if0}\ \mathbb{E}\ \mathsf{then}\ e\ \mathsf{else}\ e
\end{array}$$

Shorthands

Given $P[r \mapsto (bid, i)]$

$$\begin{array}{lll}
Bid(r) & \doteq & bid \\
Idx(r) & \doteq & i
\end{array}$$

Given $H[bid \mapsto (b, n, k)] \vee \overline{S}[bid \mapsto (b, n, k)]$
and $P[r \mapsto (bid, i)], b = \boxed{v_0, \ldots, v_{n-1}}$

$$\begin{array}{lll}
Begin(r) & \doteq & bid \\
End(r) & \doteq & bid + n \\
Size(r) & \doteq & n \\
Time(bid) & \doteq & k \\
Block(r) & \doteq & b
\end{array}$$

Given $H[bid \mapsto (b, n, k)] \vee \overline{S}[bid \mapsto (b, n, k)]$
and $P[r \mapsto (bid, i)], b = \boxed{v_0, \ldots, v_{n-1}}$
and $Begin(r) \leq r < End(r)$

$$Value(r) \doteq v_i$$

$$bid \text{ fresh} \doteq bid \notin Dom(H) \wedge bid \notin Dom(F) \wedge bid \notin Dom(\overline{S})$$

$$popStack(S, F) \doteq F \cup \Big(\bigcup_{bid \in dom(S)} (bid, s) \Big)$$

$$orig(v, f) \doteq \begin{cases} gen, & \text{if } v \text{ is a const. } n \\ z, & \text{if, } v \text{ is a gvar. } \mathsf{z} \\ f, & \text{otherwise} \end{cases}$$

Figure 3.8: Definitions and shorthands for operational semantics.

3.2.2 Operational Semantics

The operational semantics consists of state and reduction rules. The semantics is small-step, and evaluation rules have the form:

$$\langle H; F; \overline{S}; P; k; \sigma; f; e\rangle \longrightarrow \langle H'; F'; \overline{S}'; P'; k'; \sigma'; f'; e'\rangle$$

which means expression e reduces in one step to expression e', and in the process of reduction, the heap H changes to H', the freed blocks set F changes to F', the stack $\overline{S}$ changes to $\overline{S}'$, the pointer map P changes to P', the timestamp changes from k to k', the trace changes from σ to σ' and the value origin f changes to f'. We now provide definitions for state elements and then present the reduction rules.

Definitions. In Figure 3.8 we present the semantics and some auxiliary definitions. In our memory model, memory blocks b of size n are allocated in the heap via $\mathsf{malloc}\ n$ or on the stack via $\mathsf{salloc}\ n$. Block id's bid are keys in the domain of the heap or the stack; we denote their domain Bid, and represent elements in Bid as $\mathit{1}_H, \mathit{2}_H, \mathit{3}_H, \ldots$ (which indicates heap-allocated blocks) and $\mathit{1}_S, \mathit{2}_S, \mathit{3}_S, \ldots$ (which indicates stack-allocated blocks). Memory blocks are manually deallocated from the heap via $\mathsf{free}\ r$ and automatically from the stack when a function returns (the redex is $\mathsf{ret}\ \mathsf{z}\ v$). All the deallocated heap and stack blocks are stored in F—the "freed" set—as (bid, h) and (bid, s) respectively. Block contents b are represented at byte granularity, i.e., $\boxed{v_0, \ldots, v_{n-1}}$; a freshly-allocated block is not initialized, and is marked as $\boxed{\text{junk}}$. The heap H contains mappings from block id's bid to tuples (b, n, k); tuples represent the block contents b, the block size n and the timestamp k when the block was created. A stack frame S consists of mappings $\mathit{bid} \mapsto (b, n, k)$, just like the heap. The stack $\overline{S}$ is a sequence of stack frames.

We keep a pointer map P with entries $r \mapsto (bid, n)$, that is, a map from references to block id bid and offset n. Timestamps k are integers, incremented after each step. The trace σ records timed events ν, i.e., (timestamp, event) pairs. Events ν can be memory writes write, r, v, f which indicate that value v, whose origin was f, was written to location r; **if**-conditions $n == n'$ which indicate that the value of the **if** guard n was n', allocations malloc, n, bid, deallocations free, r, bid, function calls call, z, v and function return ret, z, v. At each step we keep a value origin f that tracks where the last value v comes from: a constant, a global variable, or the prior step(s), as explained shortly. Runtime expressions e are the expressions defined in Figure 3.7.

We use several notational shorthands to simplify the definition of the rules; they are shown in the right part of Figure 3.8. Given a pointer r, we can look it up in the heap H or stack $\overline{S}$, extract its bid and index i, and contents $\boxed{v_0, \ldots, v_{n-1}}$. We now explain the shorthands: $Bid(r)$ is the block id; $Idx(r)$ is the pointer's offset; $Begin(r)$ is the beginning address of a block r refers to; $End(r)$ is the end address of a block; $Size(r)$ is the size of the block; $Time(bid)$ is the timestamp at which the block was allocated; $Block(r)$ is the whole block contents; $Value(r)$ is the value stored in the memory unit pointed to by r; "bid fresh" means the bid is not in the domain of H, F, and $\overline{S}$, and bid has never been used before; $popStack(S, F)$ is used to deallocate all the blocks in the stack S, i.e., for all $bid \in dom(S)$, add (bid, s) to F.

We define the origin of a value v, denoted $orig(v, f)$, as follows: given a prior origin f, if v is a constant n, then the origin of value v is gen (value v is newly *generated* here); if v is a variable z, then the origin of value v is z (value v is propagated from variable z); otherwise, the origin of value v is f, i.e., the prior origin, indicating it is the result of a prior computation. This origin information is instrumental for constructing bug locators, as it helps track value propagation and hence bug root causes.

We use evaluation contexts $\mathbb{E}$ to indicate where evaluation is to take place next; they are modeled after expressions (shown in the left bottom of Figure 3.8), and allow us to keep reduction rule definitions simple.

Evaluation rules. The reduction rules are shown in the top of Figure 3.9. The rule [LET] is standard: when reducing $\mathsf{let}\ x = v\ \mathsf{in}\ e$, we perform the substitution $e[x/v]$. The rule [LET-SALLOC] is used to model the introduction of local variables and function arguments; it is a bit more complicated, as it does several things: first it allocates a new block *bid* of size n on the stack, initialized to junk, then it picks a fresh r and makes it point to the newly allocated block *bid* and index 0, and finally substitutes all occurrences of x with r. The allocation rule, [MALLOC], is similar: we model allocating n bytes by picking a fresh *bid*, adding the mapping $[\mathit{bid} \mapsto (\boxed{\text{junk}}, n, k)$ to the heap, creating a fresh pointer r that points to the newly-allocated block at offset 0, recording the event $(k, \mathsf{malloc}, n, \mathit{bid})$ in the trace σ, and updating the f to *gen*, meaning r is newly generated at this step. The deallocation rule, [FREE], works as follows: we first identify the *bid* that r points to, and then remove the $\mathit{bid} \mapsto (b, n, k_1)$ mapping from the heap, and add the (bid, h) tuple to F; we record the event by adding $(k, \mathsf{free}, r, \mathit{bid})$ to the trace.

The function call rule, [CALL], works as follows: create an empty stack frame S and push it onto the stack, then rewrite z v to be $\mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ (x := v; e)$, which means we allocate a new block for the function argument x on the stack, and set up the next reductions to assign (propagate) the value v to x, and then evaluate the function body e; we record the call by adding (k, call, z, v) to the trace, and propagate v's origin; we assume each function body e contains a return expression $\mathsf{ret\ z}\ e'$. The converse rule, [RETURN], applies when the next expression is a return marker; it pops the current

Evaluation

[LET]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{let}\ x = v\ \mathsf{in}\ e\rangle \longrightarrow \langle H; F; \overline{S}; P; k+1; \sigma; f; e[x/v]\rangle$	
[LET-SALLOC]	$\langle H; F; \overline{S}, S; P; k; \sigma; f; \mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ e\rangle \longrightarrow$ $\langle H; F; \overline{S}, S[bid \mapsto (\boxed{\text{junk}}, n, k)]; P[r \mapsto (bid, 0)]; k+1; \sigma; f; e[x/r]\rangle$	$r \notin Dom(P)$ $\wedge\ bid$ fresh
[MALLOC]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{malloc}\ n\rangle \longrightarrow$ $\langle H[bid \mapsto (\boxed{\text{junk}}, n, k)]; F; \overline{S}; P[r \mapsto (bid, 0)]; k+1;$ $\sigma, (k, \mathsf{malloc}, n, bid); gen; r\rangle$	$r \notin Dom(P)$ $\wedge\ bid$ fresh
[FREE]	$\langle H \uplus bid \mapsto (b, n, k_1); F; \overline{S}; P[r \mapsto (bid, 0)]; k; \sigma; f; \mathsf{free}\ r\rangle \longrightarrow$ $\langle H; F \cup (bid, h); \overline{S}; P; k+1; \sigma, (k, \mathsf{free}, r, bid); f; 0\rangle$	
[CALL]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{z}\ v\rangle \longrightarrow$ $\langle H; F; \overline{S}, S; P; k+1; \sigma, (k, \mathsf{call}, z, v); orig(v, f);$ $\mathsf{let}\ x = \mathsf{salloc}\ n\ \mathsf{in}\ (x := v; e)\rangle$	$z = \lambda x.e, S = \emptyset$
[RETURN]	$\langle H; F; \overline{S}, S; P; k; \sigma; f; \mathsf{ret\ z}\ e\rangle \longrightarrow$ $\langle H; F'; \overline{S}; P; k+1; \sigma, (k, \mathsf{ret}, z, v); orig(v, f); v\rangle$	$F' = popStack(S, F)$
[READ]	$\langle H; F; \overline{S}; P; k; \sigma; f; *r\rangle \longrightarrow \langle H; F; \overline{S}; P; k+1; \sigma; r; v\rangle$	$Value(r) = v \wedge v \neq junk$
[ASSIGN]	$\langle H[bid \mapsto (b, n, k_1]; F; \overline{S}[bid \mapsto (b, n, k_1]; P[r \mapsto (bid, i)]; k; \sigma; f; r := v\rangle$ $\longrightarrow \langle H[bid \mapsto (b', n, k_1]; F; \overline{S}[bid \mapsto (b', n, k_1]; P; k+1;$ $\sigma, (k, \mathsf{write}, r, v, orig(v, f)); orig(v, f); v\rangle$	$b' = b[i \mapsto v]$ $\wedge\ v \neq junk$
[INT-OP]	$\langle H; F; \overline{S}; P; k; \sigma; f; n_1 + n_2\rangle \longrightarrow \langle H; F; \overline{S}; P; k+1; \sigma; gen; n_3\rangle$	$n_3 = n_1 + n_2$
[PTR-ARITH]	$\langle H; F; \overline{S}; P; k; \sigma; f; r +_p n\rangle \longrightarrow$ $\langle H; F; \overline{S}; P[r_2 \mapsto (bid, i+n)]; k+1; \sigma; gen; r_2\rangle$	$Bid(r) = bid,$ $Idx(r) = i, r_2 \notin Dom(P)$
[IF-T]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{if0}\ n\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow$ $\langle H; F; \overline{S}; P; k+1; \sigma, (k, n == 0); f; e_1\rangle$	$n = 0$
[IF-F]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathsf{if0}\ n'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\rangle \longrightarrow$ $\langle H; F; \overline{S}; P; k+1; \sigma, (k, n == n'); f; e_2\rangle$	$n' \neq 0$
[CONG]	$\langle H; F; \overline{S}; P; k; \sigma; f; \mathbb{E}[e]\rangle \longrightarrow \langle H'; F'; \overline{S}'; P'; k'; \sigma'; f'; \mathbb{E}[e']\rangle$	$\langle H; F; \overline{S}; P; k; \sigma; f; e\rangle \longrightarrow$ $\langle H'; F'; \overline{S}'; P'; k'; \sigma'; f'; e'\rangle$

Error rules

[BUG-UNMATCHED-FREE]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; \mathsf{free}\ r\rangle \longrightarrow Error$	$(bid \notin Dom(H)$ $\wedge (bid, h) \notin Dom(F))$ $\vee\ r \neq Begin(r)$
[BUG-DOUBLE-FREE]	$\langle H; F; \overline{S}; P[r \mapsto (bid, 0)]; k; \sigma; f; \mathsf{free}\ r\rangle \longrightarrow Error$	$(bid, h) \in Dom(F)$
[BUG-DANG-PTR-DEREF]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; *r\rangle \longrightarrow Error$	$(bid, h) \in Dom(F)$
[BUG-DANG-PTR-DEREF2]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; r := v\rangle \longrightarrow Error$	$(bid, h) \in Dom(F)$
[BUG-NULL-PTR-DEREF]	$\langle H; F; \overline{S}; P; k; \sigma; f; *r\rangle \longrightarrow Error$	$r = NULL$
[BUG-NULL-PTR-DEREF2]	$\langle H; F; \overline{S}; P; k; \sigma; f; r := v\rangle \longrightarrow Error$	$r = NULL$
[BUG-OVERFLOW]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; *r\rangle \longrightarrow Error$	$bid \in Dom(H) \wedge$ $(r < Begin(r) \vee r \geq End(r))$
[BUG-OVERFLOW2]	$\langle H; F; \overline{S}; P[r \mapsto (bid, j)]; k; \sigma; f; r := v\rangle \longrightarrow Error$	$bid \in Dom(H) \wedge$ $(r < Begin(r) \vee r \geq End(r))$
[BUG-UNINITIALIZED]	$\langle H; F; \overline{S}; P; k; \sigma; f; *r\rangle \longrightarrow Error$	$Value(r) = junk$

Figure 3.9: Operational semantics (abstract machine states and reductions).

frame S off the stack, deallocates all the blocks allocated in S before, record the return by adding (k, ret, z, v) to the trace, and updates the f to $orig(v, f)$.

Dereferencing, modeled by the rule [READ], entails returning the value pointed to by r, and updating the f to be r, denoting that the origin of value v comes from r. When assigning value v to the location pointed to by r (which resides at block id *bid* and index i), modeled by the rule [ASSIGN], we change the mapping in the heap or stack (whichever r points to) to b', that is the block contents value at index i is replaced by v; we also record the write by adding $(k, \mathsf{write}, r, v, orig(v, f))$ to the trace, and record the assignment-induced value propagation by setting f to $orig(v, f)$.

Integer arithmetic ([INT-OP]) does the calculation, and updates the f to *gen* to mark the fact that n_3 is newly generated here; actually this rule is only necessary for purposes of value propagation, as most components of Σ remain unchanged. Pointer arithmetic ([PTR-ARITH]) is a bit more convoluted: we first find out *bid* and i—the block id and index associated with r, create a fresh r_2 that now points to block *bid* and index $i + n$ and add it to P and finally update the f to *gen*, to record that r_2 is newly generated here.

The conditional rules [IF-T] and [IF-F] are standard, though we record the predicate value and timestamp, i.e., $(k, n == 0)$ and $(k, n! = n')$, respectively, into the trace; predicate values serve as a further programmer aid. The congruence rule, [CONG], chooses where computation is to be applied next, based on the shape of $\mathbb{E}$.

Error rules. The bottom of Figure 3.9 shows the error state reduction rules. When one of these rules applies, the abstract machine is about to enter an error state—in our implementation, the debugger pauses the execution (breakpoint) just before entering an error state. These rules are instrumental for proving soundness (Appendix A) as they

```
define  Allocated(r)  =  exists event(_, malloc, _, bid)  in Trace suchthat (bid == Bid(r))
define  Freed(r, r1)  =  exists event(_, free, r1, bid)  in Trace suchthat (bid == Bid(r))
[double_free] detect <Trace; free r>:     Allocated(r) && Freed(r, r1)     :VPC(r), VPC(r1)
```

Figure 3.10: Actual bug specification input for double-free bugs.

indicate when bug detectors should fire. For brevity, we only define error rules and prove soundness for the bugs in Figure 3.2. We now proceed to describing the error rules. [BUG-UNMATCHED-FREE] indicates an illegal free r is attempted, i.e., r does not point to the begin of a legally allocated heap block. [BUG-DOUBLE-FREE] indicates an attempt to call free r a second time, i.e., the block pointed to by r has already been freed. [BUG-DANG-PTR-DEREF] and [BUG-DANG-PTR-DEREF2] indicate attempts to dereference a pointer (for reading and writing, respectively) in an already-freed block. Similarly, [BUG-NULL-PTR-DEREF] and [BUG-NULL-PTR-DEREF2] indicate attempts to dereference (read from/write to) a null pointer. Rules [BUG-OVERFLOW] and [BUG-OVERFLOW2] indicate attempts to access values outside of a block. Rule [BUG-UNINITIALIZED] applies when attempting to read values inside an uninitialized block (allocated, but not yet written to).

3.2.3 Soundness

Intuitively, our soundness property states that bug detectors fire when the underlying machine is about to enter an error state. The proof can be found in Appendix A.

3.3 Implementation

We now describe our implementation; it consists of an offline translation part that generates the detectors and locators from a bug specification, and an online debugger that runs the program and performs detection/location.

3.3.1 Debugger Code Generation

From bug specification rules, described in Section 3.1.1, automated translation via Flex[38] and Bison[39] is used to generate a **detector** and **locator** pair. We illustrate this process using Figure 3.10 which contains the full bug specification text for double-free bugs as written by the developer.

The translator first generates two helper functions for the *Allocated* and *Freed* predicates, respectively. The *Allocated* helper function parses the tracked event trace (realized by the *state monitoring* runtime library, explained shortly) to find out whether the block associated with r is allocated in the heap. The generated detector checks whether the block pointed to by pointer r is allocated in the heap and freed later whenever the program's execution reaches the start of the *free* function.

Each generated locator computes several value propagation chains based on the bug specification. For example, as shown in Figure 3.10, two value propagation chains are computed for the two pointers (r and r1) which are used to deallocate the same memory block. Each write event $\mathsf{write}, r, \_, z$ in the captured trace represents a value propagation edge from z to r. Value propagation chains are computed by traversing the value propagation edges back starting from the error detection point, until *gen* is encountered.

3.3.2 Online Debugging

Figure 3.11 shows an overview of the online debugger. The implementation runs as two separate processes (GDB and Pin) and consists of several parts: a GDB component, that provides a command-line user interface and is responsible for interpreting the target program's debugging information; a *state monitoring* component,

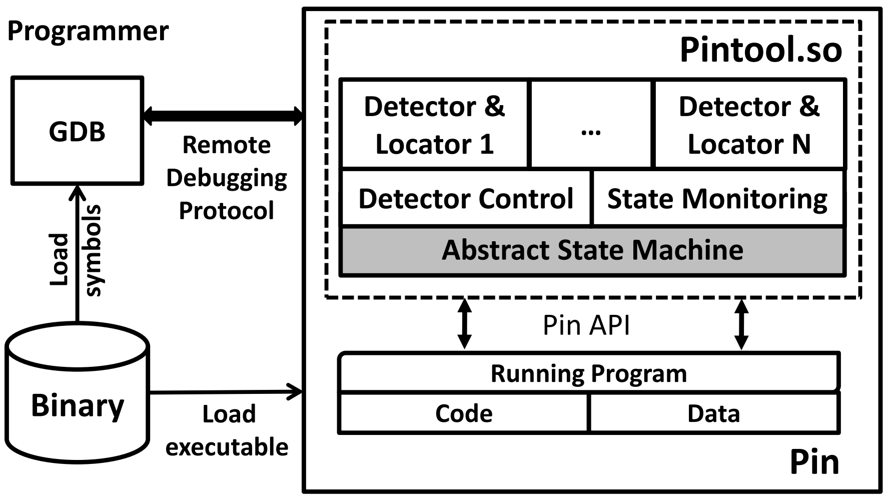


Figure 3.11: Online debugging process.

C code	Our calculus	Assembly code
		call malloc
int w;	let w_addr=salloc 4 in	$mov\ \%eax, -0x10(\%ebp)$
int **p;	let p_addr=salloc 4 in	$mov\ -0x10(\%ebp), \%eax$
p=(**int****)	p_addr:=malloc 8;	$movl\ \$0x0, (\%eax)$
malloc(4);		$mov\ -0x10(\%ebp), \%eax$
*p=0;	*p_addr:=0;	$mov\ (\%eax), \%eax$
w=**p;	w_addr:=***p_addr;	$mov\ (\%eax), \%eax$
		$mov\ \%eax, -0xc(\%ebp)$

Detection points	Tracked pointer mapping	Additions to σ
	$P[\%eax \mapsto (1_H, 0)]$	
deref_w / deref p_addr	$P[-0x10(\%ebp) \mapsto (1_H, 0)]$	(malloc, n, 1_H)
deref_r / deref p_addr	$P[\%eax \mapsto (1_H, 0)]$	(write, p_addr, _, malloc $retval$)
deref_w / deref *p_addr	$P[(\%eax) \mapsto (x, x)]$	(write, $*p\_addr$, _, gen)
deref_r / deref p_addr	$P[\%eax \mapsto (1_H, 0)]$	**bug detected at** deref $*\,p\_addr$
deref_r / deref *p_addr	$P[\%eax \mapsto (x, x)]$	

Figure 3.12: State transition for a NULL pointer dereference bug.

that tracks program execution and translates it into the abstract machine state of our calculus; and a *detector control* component that helps programmers turn detectors on and off on-the-fly. The generated bug detectors, together with the state monitoring and detector control component are linked and compiled to a pintool (a shared library) which is dynamically loaded by the Pin dynamic binary instrumentation tool. Both our state monitoring component and automatically-generated bug detectors are realized by instrumenting the appropriate x86 instructions in Pin. The GDB component communicates with the Pin-based component via GDB's remote debugging protocol.

The *detector control* module allows programmers to turn detectors on and off at runtime. When the program's execution reaches a detection point, all the detectors associated with that detection point are evaluated in the specified order. Whenever any specified bug condition is satisfied, i.e., a bug is detected, our implementation first calls PIN_ApplicationBreakpoint to generate a breakpoint at the specified statement, and then generates a bug report which consists of all the concerned events in the bug specification, as a well as the source file name and line number.

The *state monitoring* component, a runtime library, observes the program execution at assembly code level and maps it back to transitions and state changes in the abstract machine state (e.g., H, P, σ) described in Section 3.2.2. Figure 3.12 shows a NULL pointer dereference bug to illustrate how the native x86 execution is mapped to the abstract state transitions in our calculus, as well as the detection points in the detection rules. The three columns in the top half show the code in C, in our calculus, and assembly. Because C implicitly uses dereferenced pointers for stack variables (e.g., p=1 in C is really ∗(&p)=1), and our calculus makes the implicit dereference explicit, code in our calculus needs one more dereference than code in C (e.g., w:=∗∗∗p in our calculus corresponds to w=∗∗p in C). In the second column of the top of Figure 3.12 we append the *_addr* suffix to variables from the first column (e.g., *p* becomes *p_addr*) to avoid confusion.

As we can see, the x86 execution has a straightforward mapping to the state transition in our calculus. For example, the execution of the first `mov (%eax), %eax` instruction is mapped back to the [READ] evaluation rule with *r* being *p_addr* (where *r* is stored in register *eax* here), while the second `mov (%eax),%eax` is mapped back to the same rule with *r* being **p_addr* in our calculus. Meanwhile, each binary instruction has a natural mapping to the detection points (shown in the first column of the bottom half of Figure 3.12). For example, the first `mov (%eax),%eax` instruction corresponds to both *deref_r p_addr* and *deref p_addr* detection points. That is, all the bug detectors associated with *deref_r r* or *deref r* detection points are evaluated when the program is about to execute this instruction.

We generate the recording infrastructure after parsing the specifications, and only activate the required event trackers (e.g., we only activate *malloc* and *free* event trackers for double-free bugs).

Next we describe maintaining state transitions for the pointer mapping P. A block id is assigned to each allocated block, and the block id is increased after each allocation. Unique block ids ensure the detection of dangling pointer dereference bugs even when a memory block is reused. Each pointer is bound with the block id and index of the block pointed to by shadow memory. We implement the pointer mapping transition by propagating the shadow value of each pointer along with the pointer arithmetic operation. Although we only need the mapping for pointers, we temporarily maintain mapping information for registers. The second column in the bottom of Figure 3.12 shows an example of how the pointer mapping is changed by propagating the shadow value for the execution of assembly code given in the third column in the top of Figure 3.12. For example, the *malloc* function returns the address of the allocated block (e.g., the block id is 1_H) in the register *%eax*, we shadow *%eax* to (1_H, 0), denoted by $P[\%eax \mapsto (1_H, 0)]$ in Figure 3.12. The mapping info is propagated from register *%eax* into p after the execution of `mov %eax,−0x10(%ebp)`, denoted by $P[-0x10(\%ebp) \mapsto (1_H, 0)]$ in Figure 3.12, which means that pointer p points to the first element inside block 1_H. Suppose two bug detectors are generated based on the buffer overflow and NULL pointer dereference specifications in Figure 3.2. Then when the program's execution reaches the first `mov (%eax),%eax` instruction, we are at a `deref` r detection point (r is stored inside register *%eax*), and pointer mapping information for register *%eax* contains the pointer mapping information for r here($P[\%eax \mapsto (1_H, 0)]$). By evaluating the two detectors, none of the bug conditions are satisfied. The pointer mapping for register *%eax* is set to invalid (denoted by (x,x) in Figure 3.12) due to the assignment. The execution continues to the second `mov (%eax),%eax` instruction, and the NULL pointer dereference bug is reported because $r == 0$ is satisfied here (r is stored in register *%eax* and its value equals zero).

Value origin tracking is implemented similarly to pointer mapping. Each variable and register is tagged with a shadow origin of its value, and whenever the next expression to reduce is $r := v$, we update the origin (shadow value) of r to be the origin of v, and we record r and its new origin in the trace.

Storing all the tracked events and value propagations in memory may cause the debugger to run out of memory for long-running programs. Older events, which are unlikely to be accessed, can be dumped to disk and reloaded into memory if needed. However, we did not encounter this problem for our examined programs.

Bug Type	**Bug Specification** (LOC)	**Generated Debugger** (LOC)
Unmatched Free	2	2.3K
Double Free	3	2.4K
Dangling Pointer Dereference	3	2.4K
NULL Pointer Dereference	1	2.2K
Heap Buffer Overflow	3	2.3K
Uninitialized Read	1	2.2K
Total	8	3.3K

Table 3.2: Debugger code generation efficiency: comparison of lines of specification and generated debuggers for different bugs.

3.4 Experimental Evaluation

We evaluate our approach on several dimensions: *efficiency*, i.e., the manual programming effort saved by automated generation; *effectiveness/coverage*, i.e., can we (re)discover actual bugs in real-world programs?; and *performance* overhead incurred by running programs using our approach.

3.4.1 Efficiency

We measure the efficiency of our debugger code generation by comparing the lines of code of the bug specification and the generated C implementation. For each kind of bug, we specify the bug detector and bug locator as shown in Figure 3.2. Table 3.2 shows the comparison of lines of codes for bug specification and generated debugger for each kind of bug and all bugs combined.

Since detectors use the same model (detection point and predicates on the abstract machine state), and share the code for the state monitoring library, the generated code for all detectors combined is 3.3 KLOC, while for a single detector, the code size ranges from 2.2 to 2.4 KLOC. Note that the generated implementations are orders of magnitude larger than the bug specifications.

Program Name	LOC	Bug Type	Bug Location	Bug Source
Tidy-34132	35.9K	Double Free	istack.c:031	BugNet [76]
Tidy-34132	35.9K	NULL Pointer Dereference	parser.c:161	BugNet [76]
bc-1.06	17.0K	Heap Buffer Overflow	storage.c:176	BugNet [76]
Tar-1.13.25	27.1K	NULL Pointer Dereference	incremen.c:180	gnu.org/software/tar/
Cpython-870c0ef7e8a2	336.0K	Unmatched Free	typeobject.c:2490	bugs.python.org
Cpython-2.6.8	336.0K	Double Free	import.c:2843	bugs.python.org
Cpython-08135a1f3f5d	387.6K	Heap Buffer Overflow	imageop.c:593	bugs.python.org
Cpython-83d0945eea42	271.1K	NULL Pointer Dereference	_pickle.c:442	bugs.python.org

Table 3.3: Overview of benchmark programs.

3.4.2 Debugger Effectiveness

A summary of benchmarks used in our evaluation is shown in Table 3.3; each benchmark contains a real reported bug, with the details in columns 3–5. We now provide brief descriptions of the experience with using our approach to find and fix these bugs. Note that three of the bugs were presented in detail in Section 3.1.2, hence we focus on the remaining five bugs.

In addition to the double-free bug, *Tidy-34132* also contains a NULL pointer dereference which manifests when the input HTML file contains a nested *frameset*, and the *noframe* tag is unexpectedly included in the inner *frameset* rather than the outer one, which causes function *FindBody* to wrongly return a NULL pointer. *Bc-1.06* fails with a memory corruption error due to heap buffer overflow (variable *v_count* is misused because of a copy-paste error, detailed in Chapter 2). *Cpython-2.6.8* has a double-free memory bug when there is a folder in the current directory whose name is exactly the same as a module name, and this opened file is wrongly closed twice, resulting in double-freeing a *FILE* structure. *Cpython-08135a1f3f5d* crashes due to a heap buffer overflow which manifests when the *imageop* module tries to convert a very large RGB image to an 8-bit RGB. *Cpython-83d0945eea42* fails due to a NULL pointer dereference when the *_pickle* module tries to serialize a wrongly-initialized object whose *write_buf* field is NULL.

Program Name	**Traditional Debugging**	**Dynamic Slicing**	**VPC**
Tidy-34132-double-free	28,487	4,687	16
Tidy-34132-null-deref	55,777	13,050	4
bc-1.06	42,903	19,988	1
Tar-1.13.25	74	7	4
Cpython-870c0ef7e8a2	20,719	13,136	2
Cpython-2.6.8	1,083	444	10
Cpython-08135a1f3f5d	270,544	135,366	1
Cpython-83d0945eea42	11,916	7,285	2

Table 3.4: Debugging effort: instructions examined.

It can be easily seen that the benchmark suite includes bugs from our detector list and that all the bugs come from widely-used applications. Thus, this benchmark suite is representative with respect to debugging effectiveness evaluation.

All the bugs were successfully detected using the debuggers generated from the specifications in Figure 3.2. However, we did find several cases of false positives. Because our approach is based on Pin, which cannot track code execution into the kernel for system calls, our generated debuggers detected some false positives (uninitialized reads). This limitation can be overcome by capturing system call effects [74], a task we leave to future work.

We now quantify the effectiveness of our approach by showing how locators dramatically simplify the process of finding bug root causes. We have conducted the following experiment: we compute the number of instructions that would need to be examined to find the root cause of the bug in three scenarios: *traditional debugging*, *dynamic slicing*[120], and *our approach.* We present the results in Table 3.4. Traditional debugging refers to using a standard debugger, e.g., GDB, where the programmer must trace back the execution starting from the crash point to the point that represents the root cause. For the bugs considered, this would require tracing back through the execution of 74 to 270,544 instructions, depending on the program. When dynamic slicing is employed, the programmer traces back the execution along dynamic dependence edges,

Program Name	**Null Pin** seconds	**Bug Detect** seconds (factor)	**Bug Detect&VP** seconds (factor)
Tidy-34132-double-free	0.77	6.05 (7.9x)	7.62 (9.9x)
Tidy-34132-null-deref	0.62	4.52 (7.3x)	5.58 (9.0x)
bc-1.06	0.62	4.61 (7.4x)	5.70 (9.2x)
Tar-1.13.25	1.08	5.89 (5.5x)	7.43 (6.9x)
Cpython-870c0ef7e8a2	3.95	59.21 (15.0x)	80.84 (20.5x)
Cpython-2.6.8	3.31	33.16 (10.0x)	41.35 (12.5x)
Cpython-08135a1f3f5d	2.95	32.03 (10.9x)	40.13 (13.6x)
Cpython-83d0945eea42	3.17	54.21 (17.1x)	63.83 (20.1x)

Table 3.5: Execution times (from start to bug-detect), when running inside our debugger.

i.e., only a relevant subset of instructions need to be examined. Breadth-first traversal of dependence chains until the root cause is located leads to tracing back through the execution of 7 to 135,366 instructions, depending on the program. In contrast, in our approach, the programmer will trace back through the execution along value propagation chains which amounts to the examination of just 1 to 16 instructions. Hence, our approach reduces the debugging effort significantly, compared to traditional debugging and dynamic slicing.

3.4.3 Performance

The focus of our work was efficiency and effectiveness, so we have not optimized our implementation for performance. Nevertheless, we have found that the time overheads for generated monitors and locators are acceptable for interactive debugging. When measuring overhead, we used the same failing input we had used for the effectiveness evaluation. We report the results in Table 3.5. We also use the "Null Pin" running time (the program running time under Pin without our debugger/instrumentation) as the baseline, which is shown in the second column, and the time overhead with all detectors on is in the third column. The fourth column shows the time overhead with all detectors on as well as value propagation tracking on. All experiments were conducted on a DELL PowerEdge 1900 with 3.0GHz Intel Xeon processor and 3GB RAM, running Linux, kernel version 2.6.18.

From Table 3.5, we can see that the time overhead incurred by all bug detectors ranges from 5.5x to 17.1x compared to the baseline, while the time overhead incurred by all bug detectors and value propagation ranges from 6.9x to 20.5x. We believe this overhead is acceptable and a worthy tradeoff for the benefits of our approach.

When running the programs inside our debugger we have found that (1) running time increases linearly with the number of bug detectors enabled, and (2) even with the overhead imposed by our dynamic approach with all detectors and value propagation on, real-world programs took less than 81 seconds to crash on inputs that lead to bug manifestation. These results demonstrate that the overhead is acceptable and our approach appears promising for debugging tasks on realistic programs.

3.5 Summary

This chapter has presented a novel approach to constructing memory debuggers from declarative bug specifications. We have showed that many categories of memory bugs can be specified in an elegant and concise manner using First-order logic; we then prove that bug specifications are sound, i.e., they do not miss bugs that manifest during execution. We have showed that from the concise bug specifications, debuggers that catch and locate these bugs can be generated automatically, hence programmers can easily specify new kinds of bugs. We have illustrated our approach by generating debuggers for six kinds of memory bugs. Experiments with using our approach on real-world programs indicate that it is both efficient and effective.

Chapter 4

Improving the Efficiency of `Qzdb` via Input Simplification

The overhead of `Qzdb` can be quite high, because expensive runtime monitoring is needed for both high-level commands (i.e., predicate switching, dynamic slicing) and automatically generated debuggers (i.e., generated double free bug detector & locator). This is particularly the case for long program executions. This chapter tackles this problem by presenting a dynamic analysis, named *relevant input analysis*, and uses its result to enhance the *delta debugging* [112, 111, 70] algorithm for simplifying a failing program input as well as its dynamic execution with the guarantee that the same failure manifests in the simplified execution. Thus, instead of the original long execution, programmers start the debugging task with the simplified execution.

The *relevant input analysis* characterizes the *role* and *strength* of inputs in the computation of different values during a program execution. The *role* indicates whether a computed value is *derived* from an input value or its computation is simply *influenced* by an input value. The *strength* indicates if role relied upon the precise value of the input or it is among one of many values that can play a similar role. The relevant input analysis is then used to prune, as well as guide, and hence accelerate, the *delta debugging* [112, 70] algorithm.

4.1 Relevant Input Analysis

4.1.1 Motivating example

Prior relevant input analyses such as lineage tracing [114, 14, 6] identify the subset of inputs that contribute to a specified output by considering data dependence only [114], both data dependence and control dependence [14], or both data dependence and strict control dependence [6]. However, they do not characterize the role and strength of inputs in the computation of different values during execution.

An example, presented next, motivates the approach and illustrates the effectiveness of our relevant input analysis. The example is extracted from the real NULL pointer dereference bug in *Tidy-34132* (previously studied in Chapter 3). The relevant parts of the code are shown in Figure 4.1. In this simplified view, an HTML document contains a Header section (represented by 'H') and a frameset section (represented by 'S', lines 4–6). The frameset section holds one or more Frame elements (represented by 'F'), specifying the layout of views in the user agent window. In addition, the frameset section can contain a Noframes element (represented by 'N') to provide alternate content for browsers that do not support frames or have frames disabled (line 23). Noframes must contain a Body element (represented by 'B'). Framesets can be nested to any level. The body element can contain multiple Paragraphs (represented by 'P'). All the paragraphs should be included in the body element. When this property is violated, the program calls HandlePsOutsideBody (lines 51–56) to fix it. HandlePsOutsideBody sim-

```
parser.c:
void ParseHtmlDoc() {
  doc=malloc(sizeof(Doc));
  doc->seeEndBody=FALSE;
  doc->head=ParseHead();
  doc->fS=NULL;
  ParseFrameSet(NULL); }
void ParseFrameSet(Node*p)
{ Node *fS=NULL;
  char c=GetChar(fin);
  if(c=='S') {
    fS=NewNode(fSTag);
    if(p) AddChild(p,fS);
    if(doc->fS==NULL)
      doc->fS=fS;
    c=PeekChar(fin);
    while(c=='S'
          ||c=='F') {
      if(c=='S')
        ParseFrameSet(fS);
      else ParseFrame(fS);
      c=PeekChar(fin);
    }
    ParseNoFrame(fS);
    c=GetChar(fin);
    if(c=='/') ... }
}
void ParseNoFrame(Node *fS)
{ char c=GetChar(fin);
  if(c=='N') {
  Node *noF=NewNode(noFTag);
  AddChild(fS,noF);
  HandlePsOutsideBody();
  ParseBody(noF);
  HandlePsOutsideBody();
  c=GetChar(fin);
  if(c=='/') ... }
}
void ParseFrame(Node *fS)
{ char c=GetChar(fin);
  if(c=='F') {
    Node *f=NewNode(fTag);
    AddChild(fS,f); }
}
void HandlePsOutsideBody()
{ if(doc->seeEndBody==true)
  { Node *body= FindBody();
    ParseParagraphs(body); }
  else ConsumeParagraphs();
}
void ParseBody(Node *noF)
{ char c=GetChar(fin);
  if(c=='B') {
    Node *body=NewNode(bTag);
    AddChild(noF,body);
    ParseParagraphs(body);
    c=GetChar(fin);
    if(c=='/') {
      c=GetChar(fin);
      if(c=='B')
    doc->seeEndBody=true;
      else Warn(...); }
    else Warn(...); }
  else Ungetc(c,fin); }
Node *FindBody()
{ Node *node=doc->fS;
  if(node==NULL) return NULL;
  node=node->firstChild;
  while(node &&
        node->type!=noFTag)
    node=node->sibling;
  if(node) {
    node=node->firstChild;
    while(node &&
          node->type!=bTag)
      node=node->sibling; }
  return node;
}
void ParseParagraphs(Node *b)
{ char c=GetChar(fin);
  while(c=='P') { ...
    ParseTextNode(p);
    c=GetChar(fin);
    if(c=='/'){
      c=GetChar(fin);
      if(c!='P') Warn(...); }
    else Warn(...);
    c=GetChar(fin); }
  Ungetc(c,fin);
}
Node *NewNode(NodeType type)
{ Node *node=malloc(...);
    ...
  node->sibling=NULL;
  return node; }
void AddChild(Node*p,Node*c)
{ if(p>lastChild!=NULL)
    p->lastChild->sibling=c;
  else p->firstChild=c;
  p->lastChild=c;
}
void ParseTextNode(Node*p)
{ char c=GetChar(fin);
  if(c=='"') {
  c=GetChar(fin);
   ...
  c=GetChar(fin);
  if(c!='"') Warn(...); }
  else Ungetc(c,fin);
}
char GetChar(Stream *fp) {
  if(fp->r_ptr>=fp->r_end)
  return RefillBuf(fp);
  return *(fp->r_ptr++); }
```

Figure 4.1: Buggy code for illustrating relevant input analysis.

Original input:

$\boxed{S\ S\ F\ F\ N\ B\ P\ "\ a\ "\ /\ P\ /\ B\ P\ "\ b\ "\ /\ P\ /\ N\ /\ S\ /\ S}$

Inputs labeled with occurrence frequency:

$S^1\ S^2\ F^1\ F^2\ N^1\ B^1\ P^1\ "^1\ a^1\ "^2\ /^1\ P^2\ /^2\ B^2\ P^3\ "^3\ b^1\ "^4\ /^3\ P^4\ /^4\ N^2\ /^5\ S^3\ /^6\ S^4$

Compute relevant input/lineage for failure point–107_8(p is NULL):

Result of lineage [114]: {}

Result of Penumbra [14]:

$\{\} \mid \{S^1, S^2, F^1, F^2, N^1, B^1, P^1, "^1, /^1, /^2, B^2, P^3\}$

Result of lineage with strict control dependence[6]:

$\{S^1, S^2, F^1, F^2, N^1, B^1, P^1, "^1, /^1, /^2, B^2, P^3\}$

Result of our approach:

$\{S^{2=}\rightarrow$NULL(node $\rightarrow$sibling@104)$\} \wedge \{S^{1=}, S^{2=}, F^1, F^2, N^{1=}, B^{1=}, P^1, "^1, /^1, /^{2=},$
$B^{2=}\rightarrow$true(doc$\rightarrow$seeEndBody@67), $P^{3=}\} \wedge \{S^1, S^2, F^1, F^2, N^1, B^1, P^1, "^1, /^1, /^2, B^2, P^3\}$

Figure 4.2: Comparing prior work results with our relevant input analysis.

ply discards those paragraphs when a body element has not been encountered. When the end of the body has been parsed, HandlePsOutsideBody moves such paragraphs into the body element by adding all paragraphs after the body as children of body (line 53–54). The function FindBody (line 71–84) retrieves the body node. There is a bug in FindBody: the wrong assumption that noframe is always included in the outermost frameset. So when the noframe and body are included in an inner frameset, FindBody will wrongly return a NULL pointer, which causes a program crash at line 107 (p is NULL here).

Given a failure-inducing input, shown at the top of Figure 4.2), the program crashes at the eighth execution of line 107, denoted as 107_8. Consider the computation of the relevant input for variable p at failure point 107_8. The results of relevant input analyses, both as computed by prior work [114, 14, 6], as well as our algorithm, are given in Figure 4.2. As the program crashes when parsing the third P in the input, all the unprocessed inputs ("b"/P/N/S/S) are successfully excluded by all approaches. Lineage computation [114] only considers data dependence, and it gives an empty lineage because no input propagates into p at 107_8 via data dependence edges only. Penumbra [14] can be configured to consider either data dependences only, or both data and control dependences. Thus, it can generate two relevant input sets: one is empty just as lineage computation [114] or a set that includes almost all the parsed inputs. Subsequent improvements of lineage computation [6] consider both data dependences and strict control dependences, and produce the same relevant input set as Penumbra when configured considering both data and control dependences. By examining the program execution, we discover that the reason why lineage computation with strict control dependence and Penumbra include nearly all the inputs is because of the data and control dependences involving the index of buffer ($fp \rightarrow r\_ptr$) at line 124.

It is a common programming practice to maintain a buffer to store the input data and then process the data in the buffer. The program in Figure 4.1 maintains such a buffer and parses inputs based on the buffer (GetChar, Ungetc, PeekChar operate on this buffer). Hence whenever an input is read (e.g., line 86 reads the third P used in the predicate at line 87 just before the crash point), it is data dependent on the last modification of the index of the input buffer ($fp \rightarrow r\_ptr$ at line 124). Because this buffer index is increased after an input is read at line 124, and line 124 is (strict) control dependent on the predicates which guard the execution of GetChar, i.e., GetChar at line 118 is (strict) control dependent on line 114, and GetChar at line 91 is (strict) control dependent on line 87, such data and control dependence chains explain why nearly all the processed inputs are included in the relevant input. Naturally, such broad and imprecise information is not very useful.

Relevant input analysis is based upon two observations. First, data dependences incurred by operand (later defined as value dependence) should be treated differently from data dependence incurred by index or pointer (later defined as address dependence) which is used to select the operand (i.e., different dependences/inputs have different roles). Second, each dependence/input has different strength regarding the concerned output value. The relevant input analysis characterizes the *role* and *strength* that dependence/inputs play in the computation of different values during a program execution. Going back to the example in Figure 4.1, the result of our relevant input analysis is shown at the bottom of Figure 4.2. As we can see, instead of one, we present three sets: the first set includes only inputs which the concerned value p is derived from (inputs contribute to p only through value dependence); the second set includes inputs which influence p through control dependence and value dependence; and the third set includes inputs which influence p through address, control, and value dependence. Inputs labeled with $=$ in the three sets have a strong impact on the value of p. Specifically, in order to trigger or understand this bug, two conditions must be satisfied: (1) the NULL value must be generated somewhere; (2) the program execution must reach a point where this NULL value gets dereferenced. The $S^{2=}\rightarrow$NULL(node$\rightarrow$sibling@104) in our first set exactly shows that the NULL value of p is propagated from $(node \rightarrow sibling)$ at line 104, and this NULL value again is generated because of the second frameset $(S^{2=})$. The $S^{1=}$, $S^{2=}$, $N^{1=}$, $B^{1=}$, $/^{2=}$, $B^{2=}\rightarrow$true(doc$\rightarrow$seeEndBody@67), $P^{3=}$ in the second set shows that in order to cause the execution to reach this failure point, we must exactly have such inputs: $SSNB/BP$ (which turns out to be the minimal input to trigger the same bug). As we can see, our relevant input analysis provides valuable information for aiding program comprehension, debugging, test case generation, etc.

4.1.2 Definitions

Our relevant input analysis tracks dynamic dependences, originating from points where the program reads the inputs, and categorizes them to distinguish the ways in which they impact the computation of values. Given the i^{th} execution of statement s (denoted as s_i), we use $\mathsf{VAL}(sto_i)$ to denote the value computed at s_i, and during this computation, m variables are used (denoted as sfr_1, sfr_2, sfr_k, ..., sfr_m). The predicate on which s_i is control-dependent, is denoted as $pred_j$. Statement s itself can also be a predicate, in which case, $\mathsf{VAL}(sto_i)$ denotes the evaluated result of this predicate (TRUE/FALSE). We now describe the three categories.

- *Value Dependence* – $\mathsf{VAL}(sto_i)\overset{v}{\leftarrow}\mathsf{VAL}(sfr_k)$: $\mathsf{VAL}(sto_i)$ is *value dependent* upon $\mathsf{VAL}(sfr_k)$ if the latter is used as an operand for computing the former;
- *Address Dependence* – $\mathsf{VAL}(sto_i)\overset{a}{\leftarrow}\mathsf{VAL}(sfr_k)$: $\mathsf{VAL}(sto_i)$ is *address dependent* upon $\mathsf{VAL}(sfr_k)$ if the latter is used to select the address whose contents are used as an operand for computing the former. These dependences arise due to the presence of pointers and arrays; and
- *Control Dependence* – $\mathsf{VAL}(sto_i)\overset{c}{\leftarrow}\mathsf{VAL}(pred_j)$: sto_i is *dynamically control dependent* [25, 106] upon $pred_j$, i.e., $\mathsf{VAL}(pred_j)$ causes the execution of sto_i.

4.1.3 Role of Relevant Inputs

We treat all the external inputs to a program (e.g., file, stdin, network) as concerned inputs. For simplicity, we model the input as a string, and the newly-arriving inputs are simply appended to this string. The relevant inputs for a value VAL computed in a program execution that reads a set of inputs INPUTS are represented as follows:

C Code	Execution Trace	DERIVED ∧ CINFLUENCED ∧ AINFLUENCED
1 **int** data[100];	2_1 posSum=0;	VAL(posSum) ← {} ∧ {} ∧ {}
2 **int** posSum=0;	3_1 negSum=0;	VAL(negSum) ← {} ∧ {} ∧ {}
3 **int** negSum=0;	6_1 num=0;	VAL(num) ← {} ∧ {} ∧ {}
4 **int** dt;	7_1 while(!feof(fin))	VAL(!feof(fin)) ← {} ∧ {} ∧ {}
5 **int** i;	9_1 if(num>=100)	VAL(num>=100) ← {} ∧ {} ∧ {}
6 **int** num=0;	11_1 fscanf(...&dt);//3	VAL(dt) ← {3} ∧ {} ∧ {}
	12_1 data[num]=dt;	VAL(data[num]) ← {3} ∧ {} ∧ {}
7 **while**(! feof(fin))	13_1 num++;	VAL(num) ← {} ∧ {} ∧ {}
8 {	7_2 while(!feof(fin))	VAL(!feof(fin)) ← {} ∧ {} ∧ {}
	9_2 if(num>=100)	VAL(num>=100) ← {} ∧ {} ∧ {}
9 **if**(num>=100)	11_2 fscanf(...&dt);//-15	VAL(dt) ← {-15} ∧ {} ∧ {}
10 **break**;	12_2 data[num]=dt;	VAL(data[num]) ← {-15} ∧ {} ∧ {}
11 fscanf (...& dt);	13_2 num++;	VAL(num) ← {} ∧ {} ∧ {}
	7_3 while(!feof(fin))	VAL(!feof(fin)) ← {} ∧ {} ∧ {}
12 data[num]=dt;	9_3 if(num>=100)	VAL(num>=100) ← {} ∧ {} ∧ {}
	11_3 fscanf(...&dt);//0	VAL(dt) ← {0} ∧ {} ∧ {}
13 num++;	12_3 data[num]=dt;	VAL(data[num]) ← {0} ∧ {} ∧ {}
	13_3 num++;	VAL(num) ← {} ∧ {} ∧ {}
14 }	7_4 while(!feof(fin))	VAL(!feof(fin)) ← {} ∧ {} ∧ {}
15 i=0;	15_1 i=0;	VAL(i) ← {} ∧ {} ∧ {}
16 **while**(i<num)	16_1 while(i<num)	VAL(i<num) ← {} ∧ {} ∧ {}
17 {	18_1 dt=data[i];//3	VAL(dt) ← {3} ∧ {} ∧ {}
18 dt=data[i];	19_1 if(dt==0)//end?	VAL(dt==0) ← {3} ∧ {} ∧ {}
	21_1 if(dt>0)	VAL(dt>0) ← {3} ∧ {3} ∧ {}
//end marker	22_1 posSum+=dt;	VAL(posSum) ← {3} ∧ {3} ∧ {}
19 **if**(dt==0)	25_1 i++;	VAL(i) ← {} ∧ {3} ∧ {}
20 **break**;	16_2 while(i<num)	VAL(i<num) ← {} ∧ {3} ∧ {}
21 **if**(dt>0)	18_2 dt=data[i];//-15	VAL(dt) ← {-15} ∧ {3} ∧ {3}
22 posSum+=dt;	19_2 if(dt==0)	VAL(dt==0) ← {-15} ∧ {3} ∧ {3}
	21_2 if(dt>0)	VAL(dt>0) ← {-15} ∧ {3,-15} ∧ {3}
23 **else**	24_1 negSum+=dt;	VAL(negSum) ← {-15} ∧ {3,-15} ∧ {3}
24 negSum+=dt;	25_2 i++;	VAL(i) ← {} ∧ {3,-15} ∧ {3}
	16_3 while(i<num)	VAL(i<num) ← {} ∧ {3,-15} ∧ {3}
25 i++;	18_3 dt=data[i];//0	VAL(dt) ← {0} ∧ {3,-15} ∧ {3,-15}
26 }	19_3 if(dt==0)//end	VAL(dt==0) ← {0} ∧ {3,-15} ∧ {3,-15}
	20_1 break;	
27 print posSum;	27_1 print posSum;	VAL(posSum) ← {3} ∧ {3} ∧ {}
28 print negSum;	28_1 print negSum;	VAL(negSum) ← {-15} ∧ {3,-15} ∧ {3}

Figure 4.3: Example illustrating the role of input values.

$$\mathsf{VAL} \leftarrow \mathsf{DERIVED} \wedge \mathsf{CINFLUENCED} \wedge \mathsf{AINFLUENCED}$$

- Value VAL is *derived from* inputs belonging to $\mathsf{DERIVED} \subseteq \mathsf{INPUTS}$ if there is a chain of value dependences from each input in $\mathsf{DERIVED}$ to VAL:

$$\{r \mid r \in \mathsf{INPUTS} \wedge \exists\, \mathsf{VAL} \overset{v}{\leftarrow} \ldots \overset{v}{\leftarrow} \text{READ(r)}\}$$

- Value VAL is *control influenced by* inputs belonging to $\mathsf{CINFLUENCED} \subseteq \mathsf{INPUTS}$ if there is a chain of value and/or control dependences from each input in $\mathsf{CINFLUENCED}$ to VAL such that at least one control dependence is present in the chain:

$$\{r \mid r \in \mathsf{INPUTS} \wedge \exists\, \mathsf{VAL} \overset{v/c}{\longleftarrow} \ldots \overset{v/c}{\longleftarrow} \text{READ(r)}\}$$

- Value VAL is *address influenced by* inputs belonging to $\mathsf{AINFLUENCED} \subseteq \mathsf{INPUTS}$ if there is a chain of value, and/or control, and/or address dependences from each input in $\mathsf{AINFLUENCED}$ to VAL such that at least one address dependence is present in the chain:

$$\{r \mid r \in \mathsf{INPUTS} \wedge \exists\, \mathsf{VAL} \overset{v/c/a}{\longleftarrow} \ldots \overset{v/c/a}{\longleftarrow} \text{READ(r)}\}$$

We illustrate the aforementioned relevant input notions with an example in Figure 4.3 (note that we do not consider the strength of inputs for now). The code fragment on the left contains two loops. The first loop reads a sequence of numbers into the `data[]` array. The input consists of a sequence of positive and negative integers which is terminated by the value 0. The second loop scans the array and computes the sum of positive numbers (`posSum`) and sum of negative numbers (`negSum`). Finally the values of `posSum` and `negSum` are printed out. In the right column the execution trace and relevant inputs of computed values is presented for the input sequence {3, -15, 0}. The results of our analysis show that the DERIVED sets of `posSum` and `negSum` are found to be {3} and {-15} due to chains of value dependence. The CINFLUENCED set for `posSum` is {3} due to control/value dependence chain $22_1 \overset{c}{\leftarrow} 19_1 \overset{v}{\leftarrow} 18_1 \overset{v}{\leftarrow} 12_1 \overset{v}{\leftarrow} 11_1(READ(3))$. AINFLUENCED set for `posSum` is empty because no relevant inputs are propagated along address/control/value dependence chain. The CINFLUENCED set for `negSum` is {3,-15} due to chains of control/value dependences along which the values 3 and -15 are tested by predicates eventually causing the execution of statement 24_1. Note that AINFLUENCED set for `negSum` is {3} because of such address/control/value chain:

$$24_1 \overset{v}{\leftarrow} 18_2 \overset{a}{\leftarrow} 25_1 \overset{c}{\leftarrow} 19_1 \overset{v}{\leftarrow} 18_1 \overset{v}{\leftarrow} 12_1 \overset{v}{\leftarrow} 11_1(READ(3))$$

4.1.4 Strength of Relevant Inputs

Next we show that we can further qualify the inputs by determining their strength in computing other values. In particular, we determine if the computed values rely upon the *precise value* of an input, or the input value is among *one of many* values that can cause similar behavior. For this purpose a specific input value r will appear in the DERIVED, CINFLUENCED, or AINFLUENCED sets as $r^=$ (to indicate that computed value depends upon the precise value of r) or simply r (to indicate that potentially other values will lead to similar behavior as r). We now present the situations in which the above attributes can be associated when dynamic *value dependences*, *control dependences*, and *address dependences* are encountered.

Value Dependence. When the DERIVED set of a computed value VAL contains an input value $r^=$ it means that to keep VAL unchanged, we need the exact value of r (VAL is highly likely to be changed if the input value r is changed); otherwise DERIVED simply contains r (VAL may change if we change the input value r). The example below illustrates the propagation of value 10 input by the read statement. When the value 10 is first read into x and later copied to another variable y (strong value dependence), the corresponding DERIVED sets contain $10^{=}$ (strong value dependence maintains the strength of inputs). However, when the value of z is computed from the value of x at line 3 (weak value dependence), z's DERIVED set contains 10 (weak value dependence weakens the strength of inputs). Besides, because 10 has already been weakened at line 3, when the value of z is later copied to w, w contains 10 instead of $10^{=}$ (strong value dependence only maintains the strength of inputs). Similarly, when x is used in the predicate at line 5 (or 6, respectively) and it tests whether x is equal (not equal, respectively) to a precise input value 10 and when the predicate outcome is true (false, respectively), the DERIVED set will contain $10^{=}$ (strong value dependence maintains the strength of inputs); otherwise, DERIVED will simply contain 10 (line 7).

```
1: read x;             VAL(x) ← {10⁼} ∧ {} ∧ {}
2: y = x;              VAL(y) ← {10⁼} ∧ {} ∧ {}
3: z = f(x);           VAL(z) ← {10} ∧ {} ∧ {}
4: w = z;              VAL(w) ← {10} ∧ {} ∧ {}
5: if(x==10) true      VAL(x==10) ← {10⁼} ∧ {} ∧ {}
6: if(x!=10) false     VAL(x!=10) ← {10⁼} ∧ {} ∧ {}
7: if(x > 0)           VAL(x > 0) ← {10} ∧ {} ∧ {}
```

Control Dependence. If a predicate tests whether the value of a variable is equal (not equal, respectively) to a precise input value r, then the CINFLUENCED set of a statement that is control dependent upon the true (false, respectively) outcome of the predicate will contain $r^{=}$; otherwise CINFLUENCED will simply contain r. The example below illustrates the propagation of value $0^{=}$ input by the read statement and thus contained in DERIVED set of x. The value $0^{=}$ is propagated to the CINFLUENCED sets of values of w and y via control dependences.

```
1: read x;        VAL(x) ← {0=} ∧ {} ∧ {}
2: z = x;         VAL(z) ← {0=} ∧ {} ∧ {}
3: if (x==0)      VAL(x==0) ← {0=} ∧ {} ∧ {}
4:   w = z;       VAL(w) ← {0=} ∧ {0=} ∧ {}
5:   y = 1;       VAL(y) ← {0= →1(y@5)} ∧ {0=}∧{}
6: if(y < 100)    VAL(y < 100) ← {0} ∧ {0}∧{}
```

Consider a predicate that tests if an input value is precisely equal to constant c_1, and if the predicate is true, it sets another variable to a constant value c_2. Such a computation essentially maps the value of c_1 to the value c_2, i.e., c_2 is derived from c_1. Therefore in this situation we also propagate input value c_1 from the DERIVED set of a predicate to the DERIVED set of a control dependent statement that assigns c_2. The propagation also captures the mapping by including $c_1 \rightarrow c_2$ in the DERIVED set. In the above example, $0^{=}$ is propagated from DERIVED set of predicate (x==0) to the DERIVED set of y's value by inclusion of $0^{=} \rightarrow$1(y@5). Note that in such chains all values are exact values. Note that if y is later used in line 6 (weak value dependence), DERIVED and CINFLUENCED sets include 0 instead of $0^{=}$ (weak value dependence weakens the strength of inputs).

Address Dependence. If the value of a variable v used to select the address whose content (e.g., $*v$) is used as operand exactly relies on some input r, then the AINFLUENCED set of computed value VAL contain $r^{=}$ (changing the value of r will highly likely change the value of v and then $*v$); otherwise AINFLUENCED will simply contain r (changing value of r may change the value of v and then $*v$). The example below illustrates the propagation of value 10 input by the read statement. When the value 10 is first read into x and later used to select the address, the computed value z's AINFLUENCED set contains $10^=$. On the other hand, when a value of y is computed from the value of x and then used to select address, the computed value w's AINFLUENCED set contains 10. When z is tested in predicate $if(z > 0)$, the AINFLUENCED set for this predicate contains 10, rather than $10^=$.

1: read x;	$\mathrm{VAL}(x) \leftarrow \{10^=\} \wedge \{\} \wedge \{\}$
2: z = buf[x];	$\mathrm{VAL}(z) \leftarrow \{50^=\} \wedge \{\} \wedge \{10^=\}$
3: y = f(x);	$\mathrm{VAL}(y) \leftarrow \{10\} \wedge \{\} \wedge \{\}$
4: w = buf[y];	$\mathrm{VAL}(w) \leftarrow \{40^=\} \wedge \{\} \wedge \{10\}$
5: if(z > 0)	$\mathrm{VAL}(z > 0) \leftarrow \{50\} \wedge \{\} \wedge \{10\}$

4.1.5 Computation of Relevant Inputs

The dynamic value analysis is performed by instrumenting the program such that for each instruction that is executed, the relevant input sets of the computed value are found according to the dynamic dependences of the executed instruction. Figure 4.4 shows how the DERIVED (DER), CINFLUENCED (CINF) and AINFLUENCED (AINF) sets are computed via propagation of relevant input information along all dynamic dependences (Value, Address, and Control). The $\uplus$ operation in the figure is a modification of traditional union. When two values derived from same input are encountered, the stronger condition is retained:

$$\{c^=\} \uplus \{c\} = \{c^=\}$$

Similarly, when two chains are encountered such that one is a prefix of another, then the longer chain is retained as it represents a stronger condition.

```
$\textit{Initialize}$:  $\mathsf{DER}(sto_i) \leftarrow \mathsf{CINF}(sto_i) \leftarrow \mathsf{AINF}(sto_i) \leftarrow \phi$;
$\textit{Compute}$ $\mathsf{DER}(sto_i) \wedge \mathsf{CINF}(sto_i)$ $\textit{as follows:}$
for each prior statement execution on which
 $\mathsf{VAL}(sto_i)$ is directly dependent do
  – $\textit{Value Dependence}$
  case $\mathsf{VAL}(sto_i) \overset{v}{\leftarrow} \mathsf{VAL}(sfr_k)$:
    case $sto_i : \ldots = sfr_k$:
    case $sto_i : if\ (sfr_k\ ==\ c_1)$ TRUE:
    case $sto_i : if\ (sfr_k\ !=\ c_1)$ FALSE:
    $\mathsf{DER}(sto_i) \leftarrow \mathsf{DER}(sto_i) \uplus \mathsf{DER}(sfr_k)$
    $\mathsf{CINF}(sto_i) \leftarrow \mathsf{CINF}(sto_i) \uplus \mathsf{CINF}(sfr_k)$
    $\mathsf{AINF}(sto_i) \leftarrow \mathsf{AINF}(sto_i) \uplus \mathsf{AINF}(sfr_k)$
    otherwise:
    $\mathsf{DER}(sto_i) \leftarrow \mathsf{DER}(sto_i) \uplus \mathsf{DER}(sfr_k)[c^{=}\ldots/c]$
    $\mathsf{CINF}(sto_i) \leftarrow \mathsf{CINF}(sto_i) \uplus \mathsf{CINF}(sfr_k)[c^{=}\ldots/c]$
    $\mathsf{AINF}(sto_i) \leftarrow \mathsf{AINF}(sto_i) \uplus \mathsf{AINF}(sfr_k)[c^{=}\ldots/c]$
  – $\textit{Address Dependence}$
  case $\mathsf{VAL}(sto_i) \overset{a}{\leftarrow} \mathsf{VAL}(sfr_k)$:
    case $sto_i : \ldots = *sfr_k$:
    case $sto_i : *sfr_k = \ldots$:
    case $sto_i : if\ (*sfr_k\ ==\ c_1)$ TRUE:
    case $sto_i : if\ (*sfr_k\ !=\ c_1)$ FALSE:
    $\mathsf{AINF}(sto_i) \leftarrow \mathsf{AINF}(sto_i) \uplus \mathsf{DER}(sfr_k)$
      $\uplus \mathsf{CINF}(sfr_j) \uplus \mathsf{AINF}(sfr_k)$
    otherwise:
    $\mathsf{AINF}(sto_i) \leftarrow \mathsf{AINF}(sto_i) \uplus \mathsf{DER}(sfr_k)[c^{=}\ldots/c]$
      $\uplus \mathsf{CINF}(sfr_k)[c^{=}\ldots/c] \uplus \mathsf{AINF}(sfr_k)[c^{=}\ldots/c]$
  – $\textit{Control Dependence}$
  case $\mathsf{VAL}(sto_i) \overset{c}{\leftarrow} \mathsf{VAL}(pred_j)$:
    case $sto_i : \ldots = sfr_k$:
    case $sto_i : if\ (\ldots)$:
    $\mathsf{CINF}(sto_i) \leftarrow \mathsf{CINF}(sto_i) \uplus \mathsf{DER}(pred_j) \uplus \mathsf{CINF}(pred_j)$
    otherwise:
    $\mathsf{CINF}(sto_i) \leftarrow \mathsf{CINF}(sto_i) \uplus \mathsf{DER}(pred_j)[c^{=}\ldots/c]$
      $\uplus \mathsf{CINF}(pred_j)[c^{=}\ldots/c]$
    $\mathsf{DER}(sto_i) \leftarrow \mathsf{DER}(sto_i) \uplus \mathsf{CHAIN}$, such that
    case $sto_i : sto_i = c_2$ is TRUE dependent
      on $pred_j : if(var == c_1)$:
    case $sto_i : sto_i = c_2$ is FALSE dependent
      on $pred_j : if(var != c_1)$:
      CHAIN=$\{c_1^{=}\ldots \to c_2(sto_i@s) | c_1^{=}\ldots \in \mathsf{DER}(pred_j)\}$
    otherwise:  CHAIN = $\phi$
endfor
```

Figure 4.4: Dynamically computing relevant inputs of $\mathsf{VAL}(sto_i)$.

$$\{c^{=} \to d(var@s)\} \uplus \{c^{=}\} = \{c^{=} \to d(var@s)\}$$

The $S[c^{=} \ldots /c]$ operation used in Figure 4.4 is used to drop the $=$ label (i.e., weaken the strength of inputs), and it is defined as follows:

$$S[c^{=} \ldots /c] = \{c \mid c \in S \vee c^{=} \ldots \in S\}$$

For example,

$$\{c_1^{=}, c_2^{=} \to d(var@s), c_3\}[c^{=} \ldots /c] = \{c_1, c_2, c_3\}$$

In Figure 4.5 we present the results of the above analysis when it is applied to a code segment that parses a string and if the string is "`body`," then `seeBody` is set to true. The input in this case is contained in `name[]` and we assume that it is indeed the string "`body`" terminated by "\0". The first loop in the code fragment compares the input string with "`body`" which is stored in `str`. If there is an exact match, we exit the loop after setting `cmp` to 0. A chain of mappings $\backslash 0^{=}\to$0(cmp@12)$\to$BODY(tag@27)$\to$true(seeBody@29) finally leads us to statement `return seeBody` (line 30).

Both DERIVED and CINFLUENCED sets of `seeBody` at statement 30_1 capture very useful information. The chain $\backslash 0^{=}\to$0(cmp@12)$\to$BODY(tag@27)$\to$true(seeBody@29) in DERIVED set indicates how \0 is mapped to 0 for `cmp` first, and then eventually to `true` for `seeBody`. The CINFLUENCED indicates that the exact characters in "body" must be encountered as the set contains $b^{=}$, $o^{=}$, $d^{=}$, $y^{=}$, and $\backslash 0^{=}$. As we can see, such information will be very useful for program comprehension and fault localization.

C Code	Execution Trace	DERIVED ∧ CINFLUENCED ∧ AINFLUENCED
	1_1 Parse(node)	// node→name="body\0"
1 Parse(**char**∗name)	3_1 seeBody=false;	VAL(3_1)←{} ∧ {} ∧ {}

2{
 3 seeBody=false;
	4_1 str="body";	VAL(4_1)←{} ∧ {} ∧ {}

4 str="body";	6_1 i=0;	VAL(6_1)←{} ∧ {} ∧ {}
5 **int** cmp;	7_1 c=name[i];//'b'	VAL(7_1)←{$b^=$} ∧ {} ∧ {}
6 **int** i=0;	8_1 **while**(c==str[i])	VAL(8_1)←{$b^=$} ∧ {} ∧ {}
7 c=name[i];	10_1 **if** (c == '\0')	VAL(10_1)←{b} ∧ {$b^=$} ∧ {}
	15_1 i++;	VAL(15_1)←{} ∧ {b} ∧ {}
8 **while**(c==str[i])	16_1 c=name[i];//'o'	VAL(16_1)←{$o^=$} ∧ {$b^=$} ∧ {b}
9 {	8_2 **while**(c==str[i])	VAL(8_2)←{$o^=$} ∧ {$b^=$} ∧ {b}
10 **if**(c == '\0')	10_2 **if** (c == '\0')	VAL(10_2)←{o} ∧{$b^=$,$o^=$} ∧ {b}
11 {	15_2 i++;	VAL(15_2)←{} ∧ {b,o} ∧ {b}
12 cmp = 0;	16_2 c=name[i];//'d'	VAL(16_2)←{$d^=$} ∧ {$b^=$,$o^=$} ∧ {b,o}
	8_3 **while**(c==str[i])	VAL(8_3)←{$d^=$} ∧ {$b^=$,$o^=$} ∧ {b,o}
13 **break**;	10_3 **if** (c == '\0')	VAL(10_3)← {d} ∧ {$b^=$,$o^=$,$d^=$} ∧ {b,o}

14 }
 15 i++;
	15_3 i++;	VAL(15_3)←{} ∧ {b,o,d} ∧ {b,o}

	16_3 c=name[i];//'y'	VAL(16_3)←{$y^=$} ∧ {$b^=$,$o^=$,$d^=$} ∧ {b,o,d}
16 c=name[i];	8_4 **while**(c==str[i])	VAL(8_4)←{$y^=$} ∧ {$b^=$,$o^=$,$d^=$} ∧ {b,o,d}
17 }	10_4 **if** (c == '\0')	VAL(10_4) ← {y} ∧ {$b^=$,$o^=$,$d^=$,$y^=$} ∧ {b,o,d}
	15_4 i++;	VAL(15_4)←{} ∧ {b,o,d,y} ∧ {b,o,d}
18 **if**(c!=str[i])	16_4 c=name[i];//\0	VAL(16_4)←{$\backslash 0^=$} ∧ {$b^=$,$o^=$,$d^=$,$y^=$} ∧ {b,o,d,y}
19 {	8_5 **while**(c==str[i])	VAL(8_5)←{$\backslash 0^=$} ∧ {$b^=$,$o^=$,$d^=$,$y^=$} ∧ {b,o,d,y}
20 **if**(c>str[i])	10_5 **if** (c == '\0')	VAL(10_5)←{$\backslash 0^=$}
21 cmp= 1;		∧ {$b^=$,$o^=$,$d^=$,$y^=$,$\backslash 0^=$}∧ {b,o,d,y}
	12_1 cmp= 0;	VAL(12_1)←{$\backslash 0^=$→0(cmp@12)}
22 **else** 23 cmp=−1;		∧ {$b^=$,$o^=$,$d^=$,$y^=$,$\backslash 0^=$}∧ {b,o,d,y}
24 }	18_1 **if** (c!= str[i])	VAL(18_1)←{$\backslash 0^=$} ∧ {$b^=$,$o^=$,$d^=$,$y^=$} ∧ {b,o,d,y}
	25_1 tag=OTHER;	VAL(25_1)←{} ∧ {} ∧ {}
25 tag=OTHER;	26_1 **if** (cmp==0)	VAL(26_1)←{$\backslash 0^=$→0(cmp@12)}
		∧ {$b^=$,$o^=$,$d^=$,$y^=$,$\backslash 0^=$} ∧ {b,o,d,y}
26 **if**(cmp==0)	27_1 tag=BODY;	VAL(27_1)←{$\backslash 0^=$→0(cmp@12)→BODY(tag@27)}
27 tag=BODY;		∧ {$b^=$,$o^=$,$d^=$,$y^=$,$\backslash 0^=$} ∧ {b,o,d,y}
	28_1 **if** (tag==BODY)	VAL(28_1)←{$\backslash 0^=$→0(cmp@12)→BODY(tag@27)}
28 **if**(tag==BODY)		∧ {$b^=$,$o^=$,$d^=$,$y^=$,$\backslash 0^=$} ∧ {b,o,d,y}
	29_1 seeBody=true;	VAL(29_1)←{$\backslash 0^=$→0(cmp@12)→BODY(tag@27)
29 seeBody=true;		→true(seeBody@29)}
		∧ {$b^=$,$o^=$,$d^=$,$y^=$,$\backslash 0^=$} ∧ {b,o,d,y}
30 ret seeBody;	30_1 ret seeBody;	VAL(30_1)←{$\backslash 0^=$→0(cmp@12)→BODY(tag@27)
31}		→true(seeBody@29)}
		∧ {$b^=$,$o^=$,$d^=$,$y^=$,$\backslash 0^=$} ∧ {b,o,d,y}

Figure 4.5: Body parse example.

4.1.6 Implementation

We have implemented the relevant input analysis using the Pin dynamic instrumentation framework. As shown in Figure 4.4, we need to update DERIVED, CINFLUENCED and AINFLUENCED sets for each written value based on the relevant input sets of used values and the control-dependent predicate. To get more accurate control dependence, we adopted the online dynamic control dependence detection algorithm in [106]. To speed up the look-up of relevant input sets for each dependent value, we bound each computed value with its relevant input sets by shadow memory. To save space and allow efficient set operations, we stored all distinct computed relevant input sets in a balanced binary tree, and then only stored the pointer to each set in shadow memory. The CHAIN was implemented similarly to save time and space.

4.1.7 Performance Evaluation

Next we use several real programs (listed in Table 4.1) to investigate whether the time overhead of our technique is acceptable. The experiments were conducted on a machine with a 3.0GHz Intel Xeon processor and 3GB RAM, running Linux, kernel version 2.6.18. We also use the "Null Pin" time overhead as the baseline, which is shown in the second column, and the time overhead with relevant input analysis on is given in the third column. From Table 4.2, we can see that the time overhead incurred by our technique ranges from 31.7x to 39.2x compared to the baseline, which is reasonable compared to related work [114, 106].

Program	**LOC**	**Bug Source**	**Program Description**
Tidy-34132	35.9K	BugNet [76]	HTML checking & cleanup
bc-1.06	10.7K	BugNet [76]	Arbitrary-precision Calculator
Expat-1.95.3	11.9K	sourceforge.net/p/expat/bugs	XML parser

Table 4.1: Overview of benchmarks.

Program name	**Null Pin Time**	**Relevant Input Analysis Time Overhead**
	seconds	seconds (factor)
Tidy-34132	1.08	37.4 (34.6x)
bc-1.06	0.73	28.6 (39.2x)
Expat-1.95.3	0.48	15.2 (31.7x)

Table 4.2: Execution times (from start to failure point), with relevant input analysis.

4.2 Delta Debugging using Relevant Input Analysis

From the results of the preceding section it is clear that relevant input analysis can help in understanding program behavior. Therefore, in this section we show that the results of analysis can be used to develop an enhanced *delta debugging* [112, 15] algorithm. Given a program input on which the execution of a program fails, delta debugging automatically simplifies the input such that the resulting simplified input causes the same failure. In particular, it finds a 1-minimal input, i.e., an input from which removal of any entity causes the failure to disappear. This is achieved by carrying out a search in which: new simpler inputs are generated; the program is executed to determine if same failure is caused by the simpler input; and the above steps are repeatedly applied until the input cannot be simplified any further.

We now present a new delta debugging algorithm, called IDTHDD (Input Decomposition Tree-based Hierarchical Delta Debugging), that uses the result of relevant input analysis to accelerate the search for the 1-minimal input. This is achieved with the following three steps:

- **Step 1: Removal of Irrelevant Inputs.** The input is simplified by removing entities that do not appear in the relevant input set of the wrong value identifying the failure (e.g., wrong output or reference causing a crash).
- **Step 2: Construct Input Decomposition Tree**. From the dynamic dependence chop, that includes all dependence chains from input entities to faulty value, we derive a tree that represents a hierarchical decomposition of the entire input into subsets of input entities.
- **Step 3: Search for 1-Minimal Input**. The decomposition tree enables a pruned search (relative to the default delta debugging) for finding a 1-minimal input.

4.2.1 Algorithm Details

Next we will present the three steps in detail and illustrate our algorithm on the program in Figure 4.1. To better illustrate our algorithm, we use a longer failing input which has 59 entities; after removal of irrelevant inputs failing input size is 10, and finally the 1-minimal input size is 7.

Step 1: Remove Irrelevant Inputs. On a failing run, the failure is revealed either because the program crashes or it generates a wrong output. In either case, at some point in execution, a wrong value is produced and detected. Since the goal of input simplification is to reproduce the same failure, we can reduce the original input by removing all irrelevant input entities, i.e., those entities that do not appear in the relevant input set of the wrong value. We further try to reduce the input size by generating multiple inputs of different sizes from the relevant input set of the wrong value. In particular, we generate the following inputs and select the first input that reproduces the same failure. The $DER^{=}$, $CINF^{=}$ and $AINF^{=}$ sets include only those subsets of

Original input:

$\boxed{H\ ''\ t\ ''\ /\ H\ S\ F\ F\ F\ S\ F\ F\ N\ P\ ''\ a\ ''\ /\ P\ P\ ''\ b\ ''\ /\ P\ B\ P}$

$\boxed{''\ c\ ''\ /\ P\ P\ ''\ d\ ''\ /\ P\ /\ B\ P\ ''\ e\ ''\ /\ P\ P\ ''\ f\ ''\ /\ P\ /\ N\ /\ S\ /\ S}$

Inputs labeled with ocurrence frequency:
$H^1\ ''^1\ t^1\ ''^2\ /^1\ H^2\ S^1\ F^1\ F^2\ F^3\ S^2\ F^4\ F^5\ N^1\ P^1\ ''^3 a^1 ''^4 /^2 P^2\ P^3\ ''^5\ b^1\ ''^6\ /^3\ P^4\ B^1\ P^5\ ''^7\ c^1\ ''^8$
$/^4\ P^6\ P^7\ ''^9\ d^1\ ''^{10}\ /^5\ P^8\ /^6\ B^2\ P^9\ ''^{11}\ e^1\ ''^{12}\ /^7\ P^{10}\ P^{11}\ ''^{13}\ f^1\ ''^{14}\ /^8\ P^{12}\ /^9\ N^2\ /^{10}\ S^3\ /^{11}\ S^4$

Relevant inputs for 107_{14} (Failure point, p is NULL) :
VAL(107_{14}) $\leftarrow$
{$S^{2=}$→NULL(node→sibling@104)}
$\wedge$ {H^1, $''^1$, $/^1$, $S^{1=}$, $F^{1=}$, $F^{2=}$, $F^{3=}$, $S^{2=}$, F^4, F^5, $N^{1=}$, P^1, $''^3$, $/^2$, P^3, $''^5$, $/^3$, $B^{1=}$,
P^5, $''^7$, $/^4$, P^7, $''^9$, $/^5$, $/^{6=}$, $B^{2=}$→true(doc→seeEndBody@67), $P^{9=}$}
$\wedge$ { H^1, $''^1$, $/^1$, S^1, F^1, F^2, F^3, S^2, F^4, F^5, N^1, P^1, $''^3$, $/^2$, P^3, $''^5$, $/^3$, B^1, P^5, $''^7$,
$/^4$, P^7, $''^9$, $/^5$, $/^6$, B^2, P^9 }

Construct and try simpler inputs:
First input constructed from: $DER^{=}$={S^2}
$\longrightarrow$ S $\longrightarrow$ original failure cannot be reproduced.

Second input constructed from: $DER^{=} \cup CINF^{=}$= {S^1, F^1, F^2, F^3, S^2, N^1, B^1, $/^6$, B^2, P^9}
$\longrightarrow$ $S\ F\ F\ F\ S\ N\ B\ /\ B\ P$ $\longrightarrow$ **original failure is reproduced !!**

Resulting simpler input following step 1: $\longrightarrow$ $\boxed{S\ F\ F\ F\ S\ N\ B\ /\ B\ P}$

Figure 4.6: Step 1: Removing irrelevant inputs.

input entities from DER, $CINF$, and $AINF$ that are attributed with $^=$.

First Input:	$DER^=$
Second Input:	$DER^= \cup CINF^=$
Third Input:	$DER^= \cup CINF^= \cup AINF^=$
Fourth Input:	$DER \cup CINF^= \cup AINF^=$
Fifth Input:	$DER \cup CINF \cup AINF^=$
Sixth Input:	$DER \cup CINF \cup AINF$

In Figure 4.6 we show the impact of removing irrelevant inputs for our running example. The original, 59-entities input is reduced to a simple 10-entities failing input. Note that it is possible that none of the subsets can reproduce the original fault because the removal of irrelevant parts may result in a malformed input. In this case, our algorithm simply defaults to the standard delta debugging algorithm.

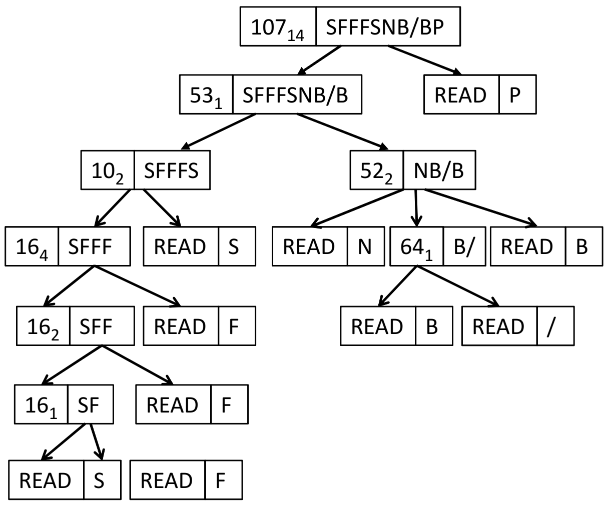


Figure 4.7: Step 2: Generating the input decomposition tree.

Step 2: Construct Input Decomposition Tree. Next, for the input obtained in the previous step, an *input decomposition tree* is constructed that hierarchically decomposes the input as follows. The root of the tree represents the wrong value computed in the failing run and its children represent a subset of other values computed during execution upon which the root value is dependent. Moreover, while the root is labeled with the entire input on which it is dependent, its children are labeled with *disjoint subsets* of inputs labeling the root node. The inputs labeling each child node of the root node are similarly further decomposed among their children and so on. Finally, each leaf node represents a read of an input value.

Thus, each level in the tree represents a decomposition of the input into disjoint subsets such that at the root node all inputs are in a single partition while at each subsequent level the inputs are decomposed into increasing number of disjoint subsets. During delta debugging, from the input decomposition at a given level in the tree, simpler inputs will be constructed by excluding or including each subset as a unit. This reduces the search space explored by delta debugging and thus accelerates the search for a 1-minimal input.

Figure 4.7 shows the input decomposition tree for our running example. As we can see, while the input associated with the root node is the entire input found in Step 1 (SFFFSNB/BP), each of the leaf nodes has a single input entity attached to it, which is the specific entity that was read by the leaf node. Internal nodes correspond to larger subsets of the input set. Note that although the leaf nodes at levels other than the last level are shown once, these nodes must be viewed as being repeated at later levels so that each level represents a decomposition of the entire input.

The input decomposition tree is derived from the dynamic dependence subgraph consisting of dynamic dependence chains originating from the input entities and terminating at the faulty value, produced as follows:

- Construct a breadth first *spanning tree* starting from the faulty value as the root and continuing until all inputs the faulty value is dependent on (i.e., inputs identified in Step 1) have been included in the tree. Collapse chains such that no node in the tree has only a single child.
- Label each node with its *input subset* which is simply the set of inputs that are reachable from the node via the edges in the spanning tree. Note that for any given level in the spanning tree, each node at that level will be labeled by a disjoint input subset since the input sets are computed using the paths that exist in the spanning tree, i.e., no input is reachable from multiple nodes at the same level in the tree.

Step 3: Search for 1-Minimal Input. We now turn to discussing how the input decomposition tree is used to search for a 1-minimal input. We apply hierarchical delta debugging according to levels in the spanning tree. At each level when delta debugging is applied, each distinct input subset at that level is viewed as a single entity, i.e., it is either entirely included in or entirely excluded from a generated input. This is similar

Level	Step	Test case		Result	
1	1	∇_{2_1}	P	$\surd$	
	2	∇_{2_2}	S F F F S N B / B	$\surd$	Go to next level
2	3	∇_{2_1}	N B / B P	$\surd$	
	4	∇_{2_2}	S F F F S P	$\surd$	Go to next level
3	5	∇_{2_1}	N B / B P	$\surd$	
	6	∇_{2_2}	S F F F S P	$\surd$	Increase granularity
	7	∇_{5_1}	S N B / B P	$\surd$	
	8	∇_{5_2}	S F F F N B / B P	$\surd$	
	9	∇_{5_3}	S F F F S B / B P	$\surd$	
	10	∇_{5_4}	S F F F S N B P	$\surd$	
	11	∇_{5_5}	S F F F S N B / P	$\surd$	Go to next level
4	12	∇_{2_1}	S N B / B P	$\surd$	
	13	∇_{2_2}	S F F F S N B P	$\surd$	Increase granularity
	14	∇_{4_1}	F S N B / B P	$\surd$	
	15	∇_{4_2}	S F F S N B / B P	$\times$	Reduce to complement
	16	∇_{3_1}	S N B / B P	$\surd$	
	17	∇_{3_2}	S F F S N / B P	$\surd$	
	18	∇_{3_3}	S F F S N B B P	$\surd$	Go to next level
5	19	∇_{2_1}	F S N B / B P	$\surd$	
	20	∇_{2_2}	S F S N B / B P	$\times$	Reduce to complement
6	21	∇_{2_1}	F S N B / B P	$\surd$	
	22	∇_{2_2}	**S S N B / B P**	$\times$	**Done - 1-minimal input found !!**

Figure 4.8: Step 3: Searching for 1-minimal input – found $\boxed{\text{S S N B / B P}}$.

to the hierarchical delta debugging [70]; although the source of hierarchy is altogether different. When applying delta debugging to each level of the input decomposition tree, we only try each complementary set instead of first trying delta sets and then complementary sets. This is based on the observation that step 1 already successfully pruned large failure irrelevant chunks from the input.

Taking the spanning tree given in Figure 4.7 as input, the input simplification using delta debugging is illustrated in Figure 4.8. We consider the root as being level 0. Therefore the figure shows inputs derived from level 1 onward. At levels 1, 2, and 3 delta debugging generates 2, 2, and 7 simpler inputs; but none of them cause a failure. Thus, we go to level 4 where the fourth simpler input reproduces the failure. This input is further simplified by applying delta debugging at level 5 which yields a simpler input that is further simplified at level 6, yielding the 1-minimal input (SSNB/BP).

As we can see, when we apply delta debugging to the input decomposition tree, for leaf nodes we have two choices: always include the leaf node in the generated input (called IDTHDD), or reconsider this leaf node again when we go to next level (called as IDTHDD*). Figure 4.8 adopts the first choice. That is, assuming we are applying delta debugging to level l in the input decomposition tree, each simpler input we try consists of two parts: the input generated by delta debugging at level l, and inputs from leaf nodes in upper levels. For example, all inputs tested at level 2 include the leaf node in level 1 ("P") in Figure 4.8. Because all nodes which read input from outside will be the leaf node in the input decomposition tree, IDTHDD* guarantees that we can get 1-minimal input. Intuitively, IDTHDD* can generate smaller (or equal) inputs with more test runs, compared with IDTHDD. However, as our experiments (discussed soon) show, IDTHDD is already good enough to get similar minimized input with much fewer test runs, compared to IDTHDD*.

4.2.2 Comparison with Standard Delta Debugging

As already shown, for our running example, the original input of size 59 is converted to a simpler 1-minimal input of size 7 and during this process the program is executed on 17 different inputs (on 2 inputs in step 1, and on 15 inputs in step 3). We now compare these results with those obtained by standard delta debugging. We found that to identify a 1-minimal input, the standard delta debugging algorithm required executing the program on 222 different inputs. Moreover, it yielded a 1-minimal input whose size is 24 (H""/HSSNPPBP""/PP""/P/BP) in contrast to the 1-minimal input of size 7 (SSNB/BP) generated by our algorithm. Thus, pruning the search space using relevant input analysis is very effective in both reducing the size of the input and the number of executions required.

We observe that removal of irrelevant inputs by Step 1 is very useful in finding smaller 1-minimal inputs. This can be explained as follows. In general, for a given original input, there may be many 1-minimal inputs that can be derived from it. The larger the original input, the more likely it is that the sizes of these 1-minimal inputs vary significantly. Since the search for 1-minimal input terminates as soon as the first such input is found, we may end up with one of the larger 1-minimal inputs when standard delta debugging is used. On the other hand, our algorithm engages in the search for a 1-minimal input only after it has eliminated the irrelevant inputs. Starting from an already simpler (i.e., smaller) input is likely to yield a smaller 1-minimal input. This is indeed what happened in the above example. After the irrelevant inputs have been removed, the input's size is 10 which is much smaller than 24, size of the 1-minimal input found by standard delta debugging.

Finally, note that finding the 1-minimal input of size 7 from the input of size 10 produced after step 1 (i.e., SFFFSNB/BP) required our algorithm to perform 15 executions of the program. On the other hand, if standard delta debugging is applied to this size 10 input (i.e., SFFFSNB/BP), it finds the same 1-minimal input as our algorithm after 37 executions of the program. Thus, guiding the search using the input decomposition tree also improves the efficiency of the search significantly (i.e., 15 vs. 37 executions).

4.2.3 Experimental Evaluation

A summary of benchmarks used in our evaluation is shown in Table 4.1; each benchmark contains a real reported bug in a widely-used program, with the details in columns 2-4. *Tidy-34132* contains a NULL pointer dereference bug. It has a similar bug trigger condition as the example in Figure 4.1: a noframe tag is included in an

Program	Test Case	DDMIN		IDTHDD		IDTHDD*	
	# chars	# test runs	# minimized input size	# test runs	# minimized input size	# test runs	# minimized input size
Tidy	2018	852	50	176	44	405	39
bc	1310	10800	191	1194	190	4185	190
Expat	1138	1785	63	216	52	393	49

Table 4.3: Summary of comparison with standard delta debugging.

Program	Test Case	Step 1		DDMIN on Simplified Input		IDTHDD -Step3	IDTHDD* -Step3
	# chars	# test runs	# simplified input	# test runs	# minimized input	# test runs	# test runs
Tidy	2018	3	124	378	44	173	402
bc	1310	2	399	8372	190	1192	4183
Expat	1138	2	125	896	56	214	391

Table 4.4: Comparison with standard delta debugging after step 1.

inner frameset and some paragraphs are wrongly placed outside body. *Bc-1.06* fails with a heap buffer overflow bug caused by a code clone error (detailed in Chapter 2). *Expat-1.95.3* fails when XML_DTD is not defined and an empty function pointer is dereferenced to allocate memory for an entity name.

Comparison with standard delta debugging. The comparison of our approach with standard delta debugging is summarized in Table 4.3. The size of original failure-inducing input is given in the second column. The number of test runs and size of minimized input for standard delta debugging is given in third and fourth column respectively. The fifth (seventh, respectively) and sixth (eighth, respectively) columns show the number of test runs and size of minimized input for IDTHDD (IDTHDD*).

As we can see, for *Tidy-34132*, IDTHDD only requires 176 test runs and produces a smaller input with size 44, while standard delta debugging needs to run 852 different inputs to produce the a minimized input with size 50. For the bug in *bc-1.06*, IDTHDD greatly outperforms standard delta debugging with *9 times* fewer test runs than standard delta debugging, while generating a slightly smaller input. For *Expat-*

1.95.3 IDTHDD generates a smaller input (52 vs. 63 characters) than standard delta debugging with much fewer test runs (8 times fewer) than standard delta debugging. For each of the three bugs, IDTHDD* generates a smaller input than IDTHDD, but requires more test runs.

To further evaluate how our relevant input analysis helps with delta debugging, the detailed comparison of our approach with standard delta debugging is presented in Table 4.4. The third and fourth columns show the number of test runs and size of simplified input for step 1 of our algorithm. The seventh (eighth, respectively) column shows the number of test runs for step 3 of IDTHDD (IDTHDD*, respectively). To show the effectiveness of input decomposition tree-based delta debugging, we also show the number of test runs and size of minimized input by applying standard delta debugging to the simplified input after step 1. As we can see, step 1 alone reduces the input size from 2018 to 124 for *Tidy-34132* (16x smaller) with 3 test runs, and from 1138 to 125 for *Expat-1.95.3* (9x smaller) with 2 test runs.

By comparing step 3 of IDTHDD with standard delta debugging, we can see that IDTHDD generates smaller (or equal) inputs with fewer test runs for the three bugs (e.g., 1192 test runs vs. 8372 runs for *bc-1.06*, and 214 test runs vs. 896 runs for *Expat-1.95.3*).

Comparison with hierarchical delta debugging. Our approach has several advantages compared to hierarchical delta debugging (HDD) [70]. First, HDD requires that the initial failure-inducing input be well-formed; otherwise, the parser which HDD is based on will fail. Note that HDD only generates syntactically valid input. However, it is common that programs often fail because of ill-formed input. For example, the original failure-inducing inputs for *Tidy-34132*, *Expat-1.95.3* and the program in Figure 4.1 are ill-formed, so HDD would fail for such kind of bugs. Second, HDD users must provide infrastructure for input parsing, unparsing a configuration, and pruning nodes from the input tree for different languages, which turns out to be non-trivial [70].

4.3 Summary

This chapter has presented a technique that greatly reduces the runtime overhead of dynamic analysis by helping the programmers focus on a much smaller input as well as a much simpler program execution while guaranteeing that the same bug is manifested as original longer failing input. To accelerate the search for a smaller input, a novel relevant input analysis is presented, which, for a particular execution, determines the role inputs play in deriving values, controlling branch predicate outcomes, and selecting referenced addresses. This information is used for delta debugging. Experiments show that relevant input analysis significantly narrows down the scope of inputs that are relevant for computing a value during execution. The benefits of narrowing the scope were demonstrated by developing an effective and efficient delta debugging algorithm.

The preceding chapters have shown how `Qzdb` overcomes the drawbacks of other general-purpose debuggers by supporting both high-level and low-level commands, providing means for extensibility as well as improved efficiency. The next chapter will show that the basic principles embodied in `Qzdb` can be extended to the debugging of multithreaded programs.

Chapter 5

The `DrDebug` Interactive Debugger for Multithreaded Programs

In this chapter we show that the approach taken by `Qzdb` can be extended to handle multithreaded programs. Cyclic debugging in the context of multithreaded programs poses multiple additional challenges:

1. Depending on the location of the bug, it can take a very long time to fast-forward and reach it.
2. Many aspects of the program state, such as heap/stack location, outcome of system calls, thread schedule, change between debugging sessions, due to thread nondeterminism.
3. Some bugs are hard to reproduce, in general and also under a debugger.

To address these additional challenges this chapter introduces a Deterministic replay based Debugging framework, or `DrDebug` for short. It is a collection of tools based on the program capture and replay framework called PinPlay [84]. PinPlay uses the Pin dynamic instrumentation system. PinPlay consists of two pintools: (i) a *logger* that

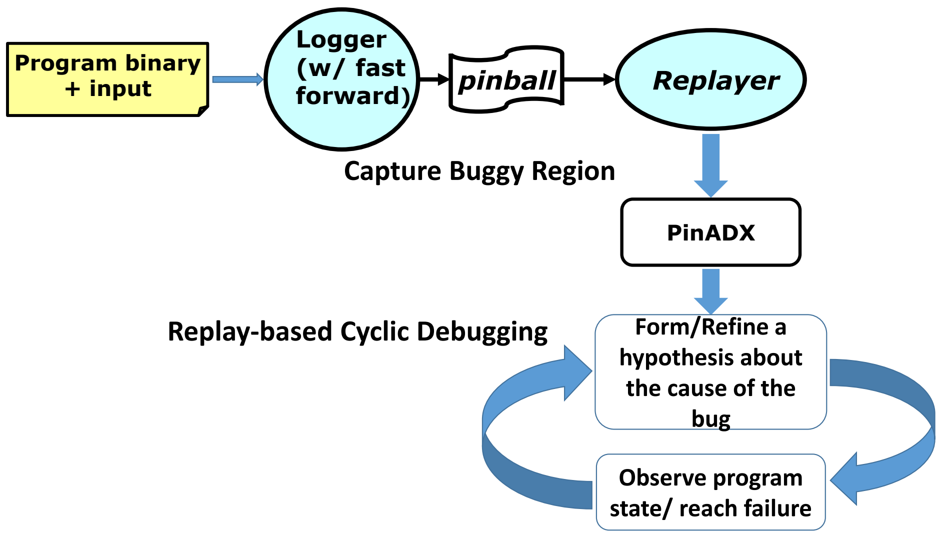

Program binary + input
Logger (w/ fast forward)
pinball
Replayer
Capture Buggy Region
PinADX
Replay-based Cyclic Debugging
Form/Refine a hypothesis about the cause of the bug
Observe program state/ reach failure

Figure 5.1: Cyclic debugging with `DrDebug`.

captures the initial architecture state and non-deterministic events during a program execution in a set of files collectively called a *pinball*; and (ii) a *replayer* that runs on a pinball repeating the captured program execution therein.

Our proposed PinPlay-based cyclic debugging process is outlined in Figure 5.1. It involves two phases: (i) capturing the buggy region in a pinball using the PinPlay logger; and (ii) replaying the pinball and using Pin's advanced debugging extension *PinADX* [61] to do cyclic debugging. Our debugger addresses various debugging challenges as follows:

1. The programmer uses the logger to fast-forward to the buggy region and then starts logging until the bug appears. Thus the generated pinball captures only an *execution region* that includes both the root-cause and the symptom of the bug. During replay-based debugging, each session starts right at the entry of the buggy region avoiding the need for fast-forwarding.

2. The programmer observes the exact same program state (heap/stack location, outcome of system calls, thread schedule, shared memory access order etc.) during

multiple debug sessions based on the replay of the same *execution region* (region pinball).

3. If the logger manages to capture a buggy pinball, it is guaranteed that the bug will be reproduced on each iteration of cyclic debugging. For hard-to-reproduce bugs, the logger can be combined with bug-exposing tools such as Maple [110] to expose and record the bug.

PinPlay also enables deterministic analysis of multithreaded programs via pintools built for analysis during replay. PinADX can make the analysis available to the user as a set of extended debugger commands. Using these two capabilities, we have designed a practical (efficient and highly precise) dynamic slicer for multithreaded programs. The dynamic slice of a computed value identifies all executed statements that directly or indirectly influence the computation of the value via dynamic data and control dependences [52]. In this chapter we greatly advance the practicality of dynamic slicing by (i) slicing execution regions to control the high cost of slicing, (ii) making a slice available across multiple debug sessions, (iii) allowing forward navigation of a slice in a live debugging session, (iv) improving its precision, and (v) handling multithreaded programs. The result is a replay debugging tool, consisting of GDB with a KDbg graphical user interface, that allows users to interactively query the statements affecting a variable value at a specific statement. Slices found once are usable across multiple debug sessions because of PinPlay's repeatability guarantee.

The key contributions of this chapter are as follows:

1. A working debugger (GDB) with a graphical user interface (KDbg) that allows deterministic cyclic debugging based on replay of pinballs for multithreaded programs. All regular debugging commands (except state modification) continue to

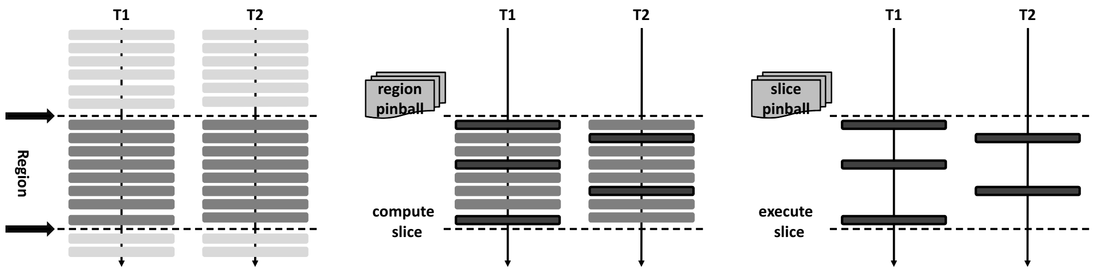
work. In addition, new commands for *execution region* (region pinball) recording and dynamic slicing are made available.

2. Handling of dynamic program slicing for multithreaded programs. The slicing works for a recorded region from a program execution. We have implemented new optimizations to make interactive slicing practical and developed analysis to make the computed slices *highly precise*.

3. Leveraging PinPlay's capabilities, we have developed a logging tool for capturing an *execution slice* which allows us to replay the execution of statements included in a dynamic slice efficiently by skipping the execution of code regions that do not belong to the slice. Programmers can load a previously generated slice and step forward from the execution of one statement in the slice to the next while examining values of program variables at each point. Such support is not provided in any prior dynamic slicing tool as they merely permit examination of slice after program execution.

4. We modified the Maple tool-chain [67] for recording the buggy executions it exposes. The resulting pinball can be readily used by `DrDebug` .

Both PinPlay and dynamic slicing can incur a large run-time overhead. However, thanks to our support for *execution region*, this overhead is incurred only within the buggy region. The overhead actually seen by the users will depend on the lengths of their buggy region. In a study of 13 buggy open source programs [77] the buggy region length (called *Window size* in the paper) was typically less than 10 million instructions, and at most 18 million instructions. In our experiment with eight 4-threaded PARSEC [8] program runs, on average, regions with 100 million instructions in the main thread (541 million instructions in all threads) could be logged in 29 seconds and replayed in 27



(a) Original execution (b) Replaying *execution region* (c) Replaying *execution slice*

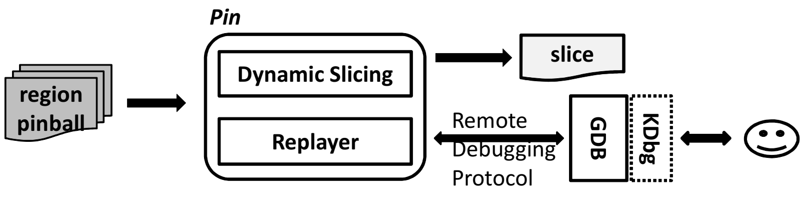
Figure 5.2: Narrowing scope of execution for replay.

seconds. We also found the overhead for region-based slicing to be quite reasonable – a few seconds to a few minutes for regions of average length 6 million instructions. Case studies of real reported concurrency bugs show the effectiveness of *execution region* and *execution slice* – the lengths of buggy *execution region* and *execution slice* are less than 15% and 7% of the total execution respectively.

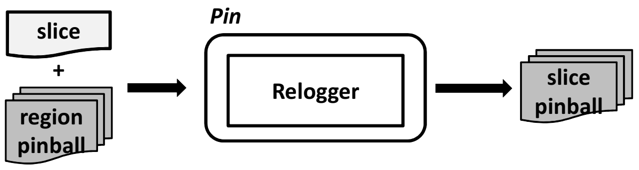
5.1 Overview of `DrDebug`

Debugging begins once a *pinball* that captures a failing run is available. This initial pinball is either generated automatically, using a testing tool, or with the assistance of the programmer. In the former case we use the Maple bug exposing tool then capture the corresponding pinball. In the latter case, we provide GDB commands/GUI buttons so the programmer can fast-forward to the buggy region and then manually capture the pinball. `DrDebug` is designed to achieve two objectives: *replay efficiency* - so that it can be used in practice; and *location efficiency* - so that the user's effort in locating the bug can be reduced.

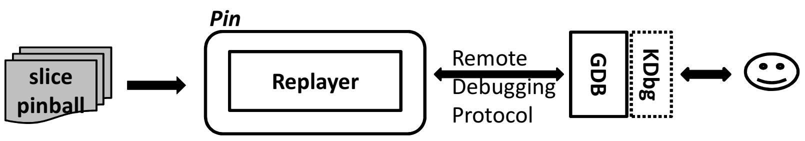
Replay efficiency. In designing `DrDebug` , one of our key objectives is to speed up debugging by increasing the speed of replay. This is particularly important for long program executions. We tackle this problem by narrowing the scope of execution that



(a) Replay execution region; compute dynamic slices.



(b) Generate slice pinball from region pinball.



(c) Replay execution slice and debug by examining state.

Figure 5.3: Dynamic slicing in `DrDebug`.

is captured by the *pinball* using the notions of *Execution Region* and *Execution Slice* in `DrDebug`.

- *Execution Region* - instead of collecting the pinball for an entire execution, users can focus on a (buggy) region of execution by specifying its start and end points. The *region pinball* is then drives replay-based debugging.

- *Execution Slice* - when studying the program behavior along a dynamic slice, instead of replaying the entire execution region, we exclude the execution of parts of the region that are not related to the slice. This is enabled by replaying using the *region pinball* and performing relogging to collect the *slice pinball.*

Figure 5.2 shows how the scope of execution that is replayed is narrowed down by the aforementioned two techniques. This greatly increases the speed of replay and makes replay based debugging practical for real world applications.

Location efficiency. To assist in locating the root cause of failure, we provide the user with a *dynamic slicing* capability. The components of the dynamic slices and their usage are (see Figure 5.3):

- When the execution of a program is replayed using the *region pinball*, our slicing pintool collects dynamic information that enables the computation of dynamic slices. Requests for dynamic slices are made by the programmer and the computed slices can be browsed or traversed going backwards along the dynamic dependences using our KDbg-based graphical user interface (see Figure 5.3(a)).
- A dynamic slice of interest found in the preceding step can be saved by the user. This slice essentially identifies a series of points in the program's execution at which the user wishes to examine the program state in greater detail. To prepare for this examination, we generate the *slice pinball* that only replays the execution of statements belonging to the slice. The *relogger* is responsible for generating the *slice pinball* from the computed *slice* by replaying using the *region pinball* (see Figure 5.3(b)).
- Finally, the user can replay the execution slice using the *slice pinball.* During this execution, breakpoints are automatically introduced allowing the user to step from the execution of one statement in the slice to the next. At each of these points, the user can examine the program state to understand program behavior (see Figure 5.3(c)).

In contrast to prior work on dynamic slicing, we make the following contributions. First, we develop a dynamic slicing algorithm that not only handles multithreaded programs, but is integrated with the replay system. Second, we provide a graphical interface which allows the user to browse a dynamic slice by traversing it backwards and examine the program state along the dynamic slice by single stepping-forward as the program executes. Finally, we have developed extensions for capturing dynamic data and control dependences that make the dynamic slice more precise. Next, we present each of these contributions in greater detail.

5.2 Computing Dynamic Slices

The dynamic slice of a computed value is defined to include the executed statements that played a role in the computation of the value. It is computed by taking the transitive closure over data and control dependences starting from the computed value and going backwards over the dynamic dependence graph. As the execution of a multithreaded program is being replayed, the user can request the computation of a dynamic slice for a computed value at any statement via our debugging interface. The steps in computing the dynamic slice are as follows:

(i) Collect Per Thread Local Execution Traces. During replay, for each thread, we collect its local execution trace that includes the memory addresses and registers defined (written) and used (read) by each instruction. This information is needed to identify dynamic dependences.

(ii) Construct the Combined Global Trace. Prior to slice computation, we combine all per-thread traces into a single fully ordered trace such that each instruction honors its dynamic data dependences including all read-after-write, write-after-write, and write-after-read dependences. The construction of this global trace requires the knowledge of shared memory access ordering to guarantee that inter-thread data dependences are also honored by the global trace. This information is already available in a pinball, as it is needed for replay. The combined global trace is thus based on the topological order of the graph, in which each execution instance is represented as a node, and an edge from node n to m means that n happens before m either in program order or in shared memory access order.

(iii) Compute Dynamic Slice by Backwards Traversing the Global Trace. A backward traversal of the global trace is carried out to recover the dynamic dependences that form the dynamic slice. We adopted the Limited Preprocessing (LP) algorithm proposed by Zhang et al. [121] to speed up the traversal of the trace. This algorithm divides the trace into blocks and by maintaining summary of downward exposed values, it allows skipping of irrelevant blocks.

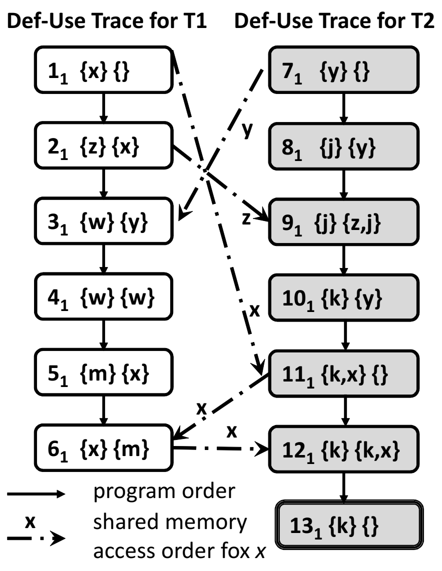
Next we illustrate our algorithm on the program in Figure 5.4. The code snippet is shown in Figure 5.4(a), where two threads, $T1$ and $T2$, operate on three shared variables (x, y, z). The code region (from line 11 to line 13) is wrongly assumed to be executed atomically in $T2$ by the programmer. However, because of the data race between statements at line 6 and line 12, x is modified unexpectedly in $T1$ by statement at line 6, causing the assertion to fail at line 13 in $T2$ (see Figure 5.4(b)). To help figure out why the assertion failed, the programmer can compute the backwards dynamic slice for k at line 13 in thread $T2$.

Figure 5.4(b) shows the individual trace for each thread. We collect the def-use information, i.e., the variables (memory locations and registers) defined and used, for each instruction. For example, 12_1 defines k by using k (defined at 10_1) and x (defined at 6_1). In addition to the per thread local traces and the shared memory access ordering used to compute the slice are also shown in Figure 5.4(b). The shared memory access orders are shown by the inter-thread dashed edges – for example, edge from 6_1 to 12_1

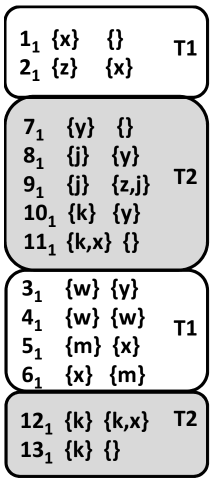
```
     T1                     T2

1    x=5;              7    y=2;
2    z=x;              8    int j=y + 1;
3    int w=y;          9    j=z + j;
4    w=w-2;            10   int k=4*y;
5    int  m=3*x;       11    if (k>x){
6    x=m+2;            12       k=k-x;
                       13      assert(k>0);
                             }
```

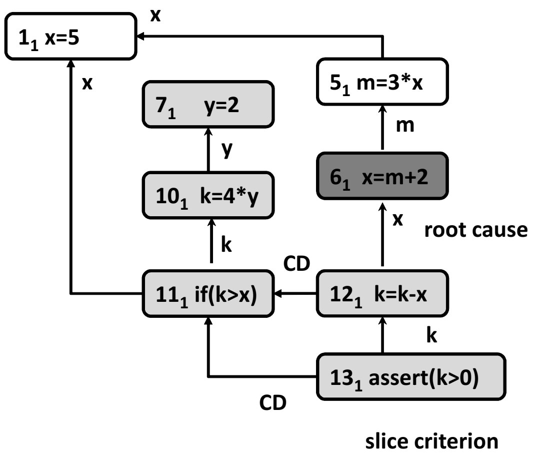
(a) Example code.



(b) Per thread traces and shared memory access order.



(c) Global trace



(d) Slice for k at 13_1.

Figure 5.4: Dynamic slicing a multithreaded program.

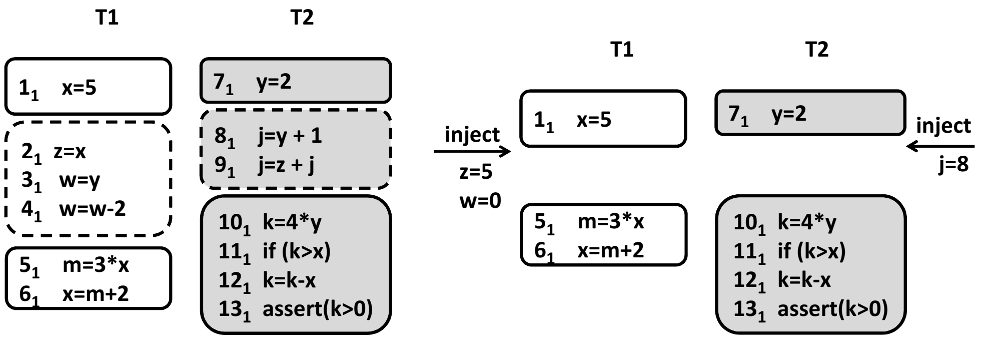
means the write of x at 6_1 in $T1$ happens before the read of x at 12_1 in $T2$. The intra-thread program orders are shown by solid edges – for example, edge $11_1 \rightarrow 12_1$ means that 11_1 happens before 12_1 in $T2$ by program order. The combined global trace for all threads shown in Figure 5.4(c) is a topological order of all the traces in Figure 5.4(b).

Using the global trace, we can then compute a backwards dynamic slice for the multithreaded program execution via a backwards traversal of the global trace to recover dependences which should be included in the slice. When we construct the global trace in step 2, we always try to cluster traces for each thread to the extent possible to improve the locality of the LP algorithm (e.g., after considering 1_1, we continue to consider 2_1 and stop at 3_1 because of the incoming edge from 7_1 to 3_1). The slice for k at 13_1 is shown in Figure 5.4(d). As we can see, the dynamic slice captures exactly the root cause of the concurrency bug: x is unexpectedly modified at 6_1 in $T1$ when $T2$ is executing an atomic region (assumed by the programmer).

Once a dynamic slice has been computed, the user can examine and navigate the slice using our graphical user interface. In addition, when an interesting slice has been found, the user may wish to engage in deeper examination of how the program state is effected by the execution of statements included in the slice as program execution proceeds. For this purpose, the user can save the slice and take advantage of replaying the execution slice as described in the next section.

5.3 Replaying Execution Slices

Prior work has used dynamic slices for postmortem analysis, after program execution, as slices identify those statement executions that influence the computation of a suspicious value via control and data dependences. However, the programmer may



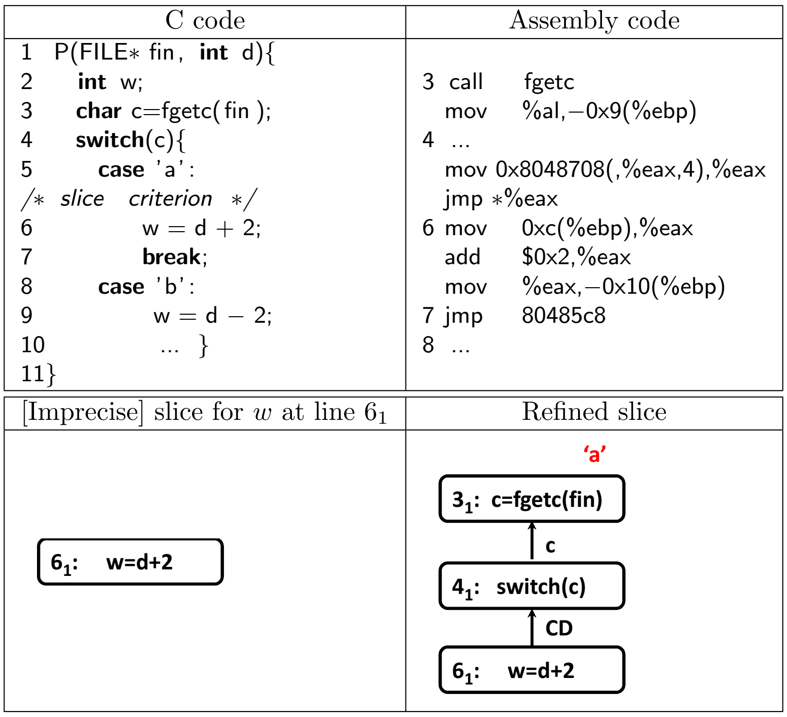
(a) Code exclusion regions. (b) Injecting values during replay.

Figure 5.5: An example of *execution slice.*

wish to examine the concrete values of variables at statement instances in the slice to see how these statements impact program state. Therefore we support the idea of *replaying an execution slice* which provides two key features.

- First, the user can examine the values computed along the slice in a live debugging session. In fact we allow the user to step through the execution of the program from one statement in the slice to the next statement in the slice.
- Second, for *efficiency*, only the part of computation that forms the slice is replayed. To implement this feature we leverage PinPlay's relogging and code exclusion features.

PinPlay's relogger can run off a pinball and then generate a new pinball by excluding some code regions. Given a slice, `DrDebug` can exclude all the code regions which are not in the slice and generate a *slice pinball.* PinPlay's relogger maintains a per-thread exclusion flag to support local exclusion regions for each thread. Given an exclusion code region $[startPc : sinstance : tid,\ endPc : einstance : tid)$ for thread tid, relogger sets the exclusion flag and turns on the side-effects detection when $sinstance^{th}$ execution of $startPc$ is encountered, and then resets the flag when the $einstance^{th}$ execution of $endPc$ is reached in thread tid.



C code	Assembly code

```
1  P(FILE* fin, int d){
2    int w;
3    char c=fgetc(fin);
4    switch(c){
5      case 'a':
/* slice criterion */
6          w = d + 2;
7          break;
8      case 'b':
9           w = d - 2;
10           ... }
11}
```

```
3  call    fgetc
   mov     %al,-0x9(%ebp)
4  ...
   mov 0x8048708(,%eax,4),%eax
   jmp *%eax
6  mov     0xc(%ebp),%eax
   add     $0x2,%eax
   mov     %eax,-0x10(%ebp)
7  jmp     80485c8
8  ...
```

[Imprecise] slice for w at line 6_1	Refined slice

Figure 5.6: Control dependences in the presence of *indirect jumps*.

To enable generation of the *slice pinball*, we output a special slice file which, in addition to the normal slice file, also identifies the exclusion code regions. As shown in Figure 5.5(a), we identify all the exclusion code regions (shown as dashed boxes) for each thread, and output such information to the special slice file. The relogger leverages this file to generate the *slice pinball*. Relogger detects the side-effects of excluded code regions using the same algorithm PinPlay adopted for system call side-effects detection [75]. When `DrDebug` runs off the *slice pinball*, all the excluded code regions will be completely skipped and their side-effects are restored by injecting modified memory cells and registers as shown in Figure 5.5(b).

5.4 Improving Dynamic Dependence Precision

The utility of a dynamic slice depends upon the precision with which dynamic dependences are computed. We observe that prior dynamic dependence detection al-

C code	Assembly code
`1 P(FILE* fin, int d){`	`3 call fgetc`
`2   int w, e;`	`  mov %al,−0x9(%ebp)`
`3   char c=fgetc(fin);`	`4 mov 0xc(%ebp),%eax`
`4   e= d + d;`	`  add %eax,%eax`
`5   if(c=='t')`	`5 cmpb $0x74,−0x9(%ebp)`
`6     Q();`	`  jne 804852d`
`/* slice criterion */`	`6 call Q`
`7   w=e;`	`  804852d:`
`8 }`	`7 mov %eax,−0x10(%ebp)`
`9 Q()`	
`10 {`	`9 Q()`
`11   ...`	`10 push %eax`
`12 }`	`...`
	`12 pop %eax`

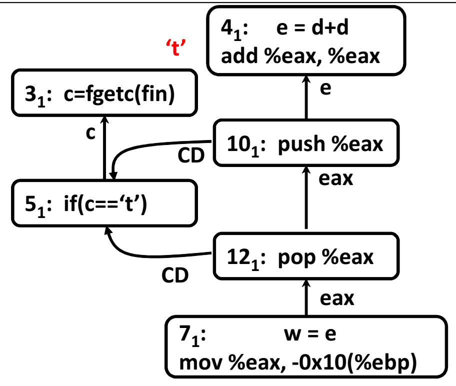
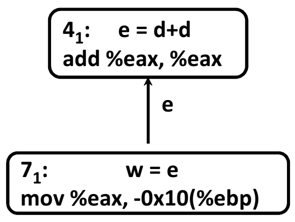
[Imprecise] slice for w at line 7_1	Refined slice
	

Figure 5.7: Spurious dependence example.

gorithms (e.g., [107, 106, 121, 117]) which leverage binary instrumentation frameworks (e.g., Pin [62], Valgrind [80]) have two sources of imprecision. First, in the presence of *indirect jumps*, these algorithms fail to detect certain dynamic control dependences causing statements to be missed from the dynamic slice. Second, due to the presence of *save* and *restore* operation pairs at function entry and exit points, spurious data dependences are detected causing the dynamic slices to be unnecessarily large. Next we address these sources of imprecision. To the best of our knowledge, we are the first to observe and propose solutions to mitigate these problems.

5.4.1 Dynamic Control Dependence Precision

For accurately capturing dynamic control dependence in the presence of recursive functions and irregular control flow, we use the online algorithm by Xin and Zhang [106]. However, using this algorithm in the context of a dynamic binary instrumentation framework poses a major challenge, as it assumes the availability of precomputed static immediate post-dominator information for each basic block. Due the presence of indirect jumps, accurate static construction of the control flow graph is not possible. As a result, the post-dominator information precomputed statically is imprecise and the dynamic control dependences computed are imprecise as well. In prior works [117, 106, 107] this problem is addressed by restricting the applicability of the slicing tool to binaries generated using a specific compiler which limits the applicability of the tool.

Let us illustrate the problem caused by an indirect jump using the example in Figure 5.6. The code snippet is shown in the top left column, and the top right column shows its assembly code. The switch-case statement is translated to the indirect jump: `jmp *%eax`. Without the dynamic jump target information, in general, the static analyzer cannot figure out the possible jump targets for a indirect jump. Thus, the statically constructed CFG will be missing control flow edges from statement 4 to statements 6 and 9. This inaccurate CFG leads to an imprecise dynamic slice shown in the bottom left column with missing control dependence $6_1 \rightarrow 4_1$.

To achieve wide applicability and precision we take the following approach in `DrDebug` . We implement a static analyzer based on Pin's static code discovery library – this allows `DrDebug` to work with any x86 or Intel64 binary. Further, we develop an algorithm to improve the accuracy of control dependence in the presence of indirect jumps. Initially we construct an approximate static CFG and as the program executes, we collect the dynamic jump targets for the indirect jumps and refine the CFG by adding the missing edges. The refined CFG is used to compute the immediate post-dominator for each basic block which is then used to dynamically detect control dependences. This leads to the accurate slice shown in the bottom right column in Figure 5.6.

5.4.2 Dynamic Data Dependence Precision

Besides memory to memory dependences, we need to maintain the dependences between registers and memory to perform dynamic slicing at the binary level. Dynamic slices may include many spurious dependence edges when registers are saved/restored upon at function entry/exit. More specifically, at each function entry, registers used inside this function are saved on the stack, and later restored from the stack in reverse order when the function returns to its caller.

Consider the example in Figure 5.7. The top left and top right columns show a C code snippet and its corresponding assembly code respectively. Register *eax* is used in function Q, and its value is saved/restored onto/from stack at line 10/12. Considering an execution where c's value is 't' at line 3, let us compute a slice for w at the first execution of statement at line 7. As variable e is used to compute w in 7_1 and its value is stored in *eax*, we continue to backwards traverse the trace to find the definition of register *eax*. As value of *eax* is saved/restored onto/from stack at the entry/exit of Q; we will establish data dependence edges $7_1 \rightarrow 12_1$, $12_1 \rightarrow 10_1$, and $10_1 \rightarrow 4_1$ due to *eax*. If only data dependences are considered, we get longer data dependence chains than needed. Since a dynamic slice is a transitive closure of both control and data dependences, we may wrongly include many spurious data and control dependences because of such data dependence chains. In the Figure 5.7 example, because all statements (e.g., 10_1 and

12_1) in function Q are directly or indirectly control dependent on predicate 5_1 which guards the execution of function Q, a slice for w at 7_1 will wrongly include 3_1 and 5_1 (as shown in the bottom left column) as well as all other statements on which 3_1 and 5_1 are dependent.

We call a pair of instructions that are only used to save/restore registers a *save/restore pair* and data/control dependences which are introduced by such pairs as *spurious dependences.* To improve the precision of the dynamic slice, we propose to precisely identify save/restore pairs and prune the spurious data/control dependence resulting from them.

Dynamically identifying save/restore pairs. One possible way is to have the compiler generate special markers for the save/restore pairs so they can be easily identified at runtime. This approach would limit the applicability of `DrDebug` since `DrDebug` is designed to work with any unmodified x86 or Intel64 binary. Therefore we use a dynamic algorithm for detecting as many save/restore pairs as possible without the help of the compiler. Our algorithm handles the following complexities caused by the compiler. First, the compilers can use either *push* (*pop*) or *mov* instruction to save (restore) the value of a register. Moreover, *push/pop* instructions are not exclusively used to save/restore registers. Second, it is not easy to know how many *push* (*pop*) or *mov* instructions are exactly used to save (restore) registers at the entry (exit) of a function. Our algorithm works as follows:

- *Statically identify potential save and restore instructions.* The first $MaxSave$ push/mov reg2mem instructions at the start of a function and the last $MaxSave$ pop/mov mem2reg instructions at the end of a function are identified as potential *save* and *restore* instructions respectively. $MaxSave$ is a tunable parameter.

- *Dynamically verify that the pairs are used to save and restore registers.* For each potential *save* instruction, we record register/memory pair and the saved value from the register. For each potential *restore* instruction, we record register/memory pairs and the restored value from the stack. An identified save/restore pair must satisfy two conditions: (1) *save* copies the value of a register r to stack location s at the entry of a function; and (2) *restore* copies the same value from s back to r at the exit of the same function. In the example of Figure 5.7, 10_1 and 12_1 are recognized as a save/restore pair for eax.

Pruning spurious data dependences. With recognized save/restore pairs, we prune spurious data dependence by bypassing data dependences caused by such save/restore pairs. Take the slice in the bottom left column in Figure 5.7 as an example. Because $7_1 \rightarrow 12_1$, $12_1 \rightarrow 10_1$, $10_1 \rightarrow 4_1$, and 10_1 and 12_1 are recognized as a save/restore pair for eax, we bypass the data dependence chain and add a direct edge $7_1 \rightarrow 4_1$. In this way, the refined slice for w at 7_1 will not include 3_1, 5_1, and all other statements on which 3_1 and 5_1 are dependent, as shown in the bottom right column.

5.5 Implementation

The implementation of `DrDebug` consists of Pin-based and GDB-based components. The Pin-based component consists of the PinPlay library (available for download [86]) and the Dynamic Slicing module. The programmer interfaces with the GDB component via a command line interface or a KDbg based graphical interface. The GDB component communicates with the Pin-based component via PinADX. PinPlay's logger is leveraged to generate a (region) pinball and then PinPlay's replayer can deterministically replay the execution for multithreaded program by running off such pinball.

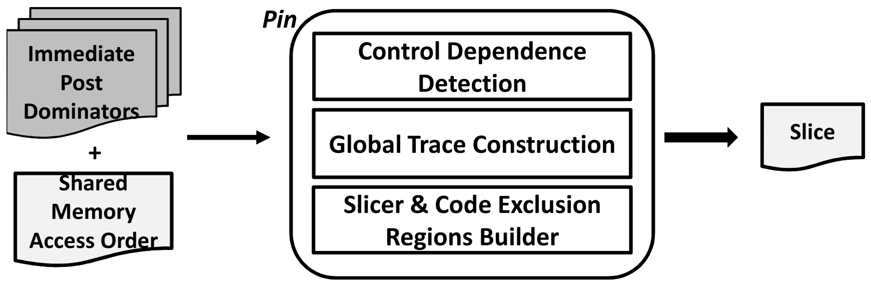
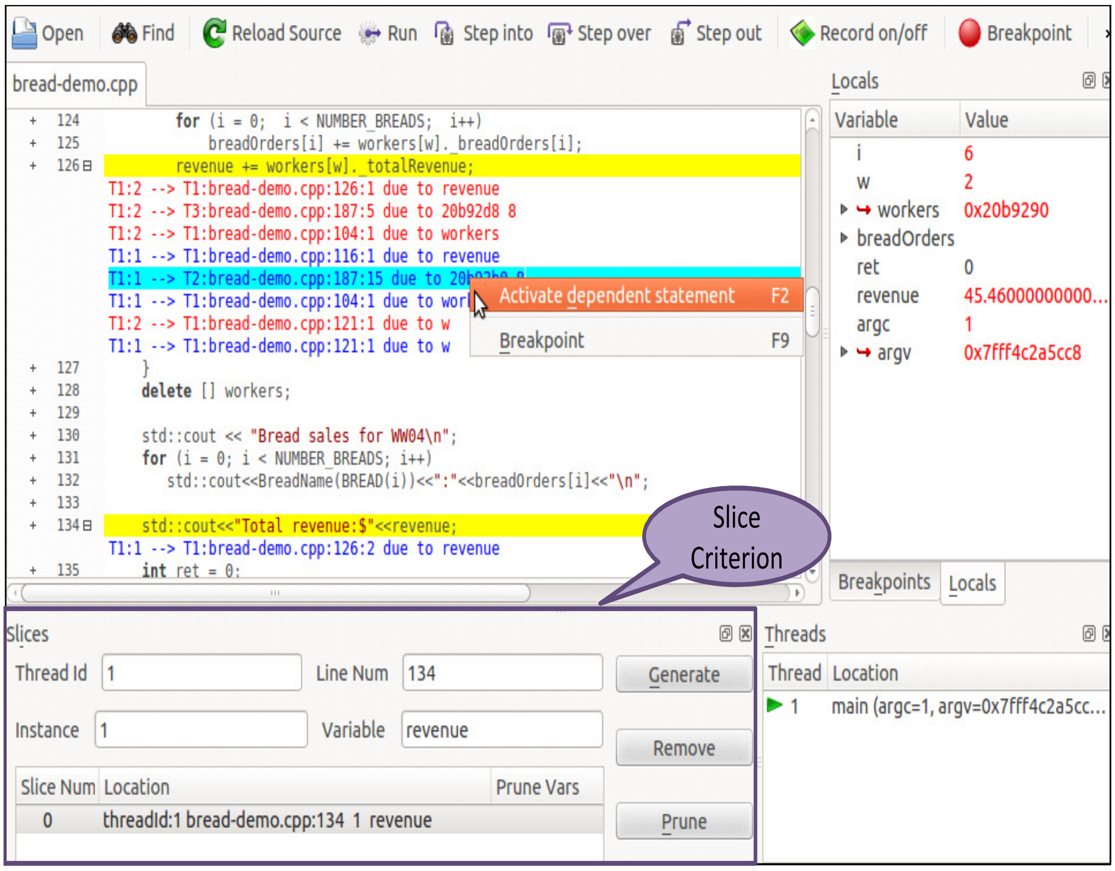

Immediate Post Dominators
+
Shared Memory Access Order
Pin
Control Dependence Detection
Global Trace Construction
Slicer & Code Exclusion Regions Builder
Slice

Figure 5.8: Dynamic slicer implementation.

During the replay, driven by a slice command from GDB-based component, the dynamic slicing module computes a slice and then PinPlay's relogger is leveraged to generate a *slice pinball*. Finally the user can single step/examine just statements in the slice when PinPlay's replayer runs off the *slice pinball*.

Dynamic Slicer. The implementation of the dynamic slicing module is shown in Figure 5.8. We implement a static analysis module based on Pin's static code discovery library to conduct analysis to generate the control flow graph and compute the immediate post dominator information. Given immediate post dominators information, the *Control Dependence Detection* submodule [106] detects the dynamic control dependences. The *Global Trace Construction* submodule tracks individual thread traces and then constructs the global trace based on the shared memory access orders which were captured by the PinPlay's logger to enable deterministic replay. When the user issues a slice command, the *Slicer & Code Exclusion Regions Builder* submodule computes the slice by backwards traversing the global trace and then outputs the slice in two forms: a normal slice file used for slice navigation and browsing in KDbg; and another slice formatted as a sequence of code exclusion regions for use by the relogger to generate the *slice pinball*.



```
+ 124         for (i = 0;  i < NUMBER_BREADS;  i++)
+ 125             breadOrders[i] += workers[w]._breadOrders[i];
+ 126         revenue += workers[w]._totalRevenue;
T1:2 --> T1:bread-demo.cpp:126:1 due to revenue
T1:2 --> T3:bread-demo.cpp:187:5 due to 20b92d8 8
T1:2 --> T1:bread-demo.cpp:104:1 due to workers
T1:1 --> T1:bread-demo.cpp:116:1 due to revenue
T1:1 --> T2:bread-demo.cpp:187:15 due to 20b
T1:1 --> T1:bread-demo.cpp:104:1 due to wor
T1:2 --> T1:bread-demo.cpp:121:1 due to w
T1:1 --> T1:bread-demo.cpp:121:1 due to w
+ 127     }
+ 128     delete [] workers;
+ 129
+ 130     std::cout << "Bread sales for WW04\n";
+ 131     for (i = 0; i < NUMBER_BREADS; i++)
+ 132        std::cout<<BreadName(BREAD(i))<<":"<<breadOrders[i]<<"\n";
+ 133
+ 134     std::cout<<"Total revenue:$"<<revenue;
T1:1 --> T1:bread-demo.cpp:126:2 due to revenue
+ 135     int ret = 0;
```

Variable	Value
i	6
w	2
workers	0x20b9290
breadOrders	
ret	0
revenue	45.46000000000...
argc	1
argv	0x7fff4c2a5cc8

Thread Id	1	Line Num	134
Instance	1	Variable	revenue

Slice Num	Location	Prune Vars
0	threadId:1 bread-demo.cpp:134 1 revenue	

Thread	Location
1	main (argc=1, argv=0x7fff4c2a5cc...

Figure 5.9: DrDebug GUI showing a dynamic slice.

GUI. The extended KDbg provides an intuitive interface for selecting interesting regions for logging, and computing, inspecting and stepping through slices during replay. Figure 5.9 shows a screen-shot of our interface – all the statements in the slice are highlighted in yellow. The programmer can access the concrete inter-thread dependences and navigate backwards along dependence edges by the clicking on the Activate button of the dependent statement.

Integration with Maple. Maple [110] is a coverage-driven testing tool-set for multithreaded programs. One of the usage models it supports helps when a programmer accidentally hits a bug for some input but is unable to reproduce the bug. Maple has two phases (i) a profiling phase where a set of inter-thread dependencies, some observed and some predicted, are recorded, and (ii) an active scheduling phase that runs the program on a single processor and controls thread execution (by changing scheduling priorities) to enforce the dependencies recorded by the profiler. The active scheduler does multiple runs until the bug is exposed.

Since Maple is based on Pin, it is an ideal candidate for integration with `DrDebug` . We changed the active scheduler pintool in Maple to optionally do PinPlay-based logging of the buggy execution it exposes. We had to make sure, using Pin's instrumentation ordering feature, that the active scheduler's thread control does not interfere with PinPlay logger's analysis.

We have successfully recorded multiple buggy executions for the example programs in the Maple distribution. The pinballs generated could be readily replayed and debugged under GDB. We have pushed the changes we made to Maple's active scheduler back to the Maple sources [67].

5.6 Experimental Evaluation

5.6.1 Case Studies

We studied 3 real concurrency bugs from three widely used multithreaded programs. The bug descriptions are detailed in the last column of Table 5.1. The case studies mainly serve two purposes: (a) quantify the execution region sizes (i.e., the number of executed instructions) that need to be logged and replayed later in order to capture and fix each bug; (b) `DrDebug` has reasonable time and space overhead for real concurrency bugs with both whole program execution (i.e., execution region from program beginning to failure point) and buggy execution region (e.g., execution region from root cause to failure point).

Program Name	**Program Description**	**Type**	**Bug Source**	**Bug Description**
PBZIP2-0.9.4	Parallel file compressor	Real	[115]	A data race on variable $fifo \rightarrow mut$ between main thread and the compressor threads.
Aget-0.57	Parallel downloader	Real	[109]	A data race on variable *bwritten* between downloader threads and the signal handler thread.
mozilla-1.9.1	Web browser	Real	[44]	A data race on variable $rt \rightarrow scriptFilenameTable$. One thread destroys a hash table, and another thread crashes in $js\_SweepScriptFilenames$ when accessing this hash table.

Table 5.1: Data race bugs used in our experiments.

Program Name	**#executed instructions**	**#instructions (%instructions) in slice pinball**	**Logging Overhead**		**Replay Time** (sec)	**Slicing Time** (sec)
			Time (sec)	**Space** (MB)		
PBZIP2	11186	1065 (9.5%)	5.7	0.7	1.5	0.01
Aget	108695	51278(47.2%)	8.4	0.6	3.9	0.02
mozilla	999997	100 (0.01%)	9.9	1.1	3.6	1.2

Table 5.2: Time and Space overhead for data race bugs with buggy execution region.

Program Name	**#executed instructions**	**#instructions (%instructions) in slice pinball**	**Logging Overhead**		**Replay Time** (sec)	**Slicing Time** (sec)
			Time (sec)	**Space** (MB)		
PBZIP2	30260300	11152 (0.04%)	12.5	1.3	8.2	1.6
Aget	761592	79794 (10.5%)	10.5	1.0	10.1	52.6
mozilla	8180858	813496 (9.9%)	21.0	2.1	19.6	3200.4

Table 5.3: Time and space overhead for data race bugs with whole program execution region.

Table 5.2 shows the time and space overhead with buggy execution region for each bug. For each bug, we captured the execution from the root cause to the failure point, and then computed a slice for the failure point during deterministic replay. The number of executed instructions is shown in the second column, while the number of instructions captured in the slice pinball, as well as the percentage of number of instructions in the slice pinball over total executed instruction, are presented in the third column. The time and space overhead for logging is shown in the fourth and fifth column respectively. The sixth column shows the time to replay the captured buggy region pinball, and the time for slicing is shown in the seventh column. As we can see, all concurrency bugs we studied can be reproduced with region size of 1 million instructions. Besides, the time overhead for logging, replay, and slicing is reasonable.

The time and space overhead with whole execution for each bug is shown in Table 5.3. For each bug, we captured the execution from the beginning of program to the failure point, simulating that novice programmers tend to capture large execution regions. As we can see, all concurrency bugs can be reproduced from the program beginning, with maximal region size of 31 million instructions. The logging, replay, and slicing time overhead is acceptable, considering the large amount of time programmers spend on debugging.

5.6.2 Logging and Replay

We first present results from log/replay time evaluations using 64-bit pre-built binaries with suffix 'pre' for version 2.1 of the PARSEC [8] benchmarks run on the *native* input. The goal of our evaluations was to find the logging/replay time for regions of varying sizes. We first evaluated the 4-threaded runs to find a region in the program where all four threads are created. We then chose an appropriate *skip* count for the

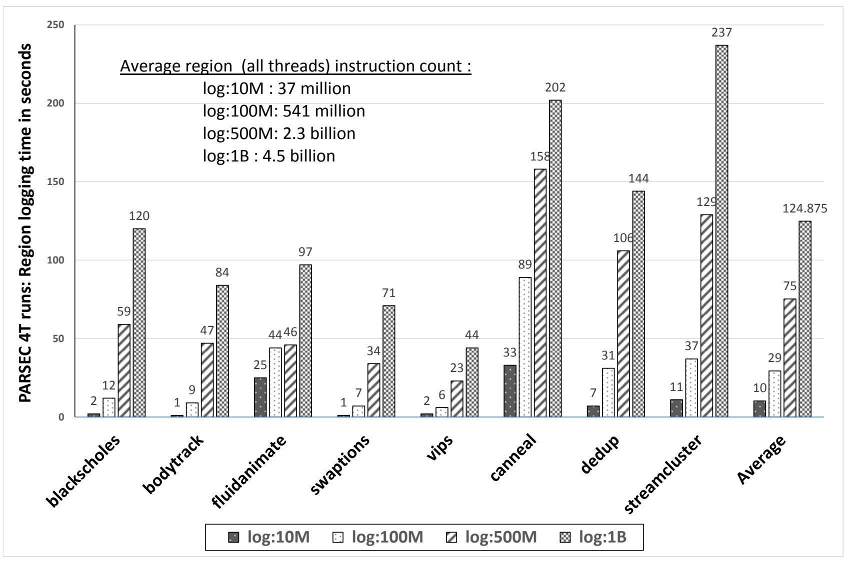


Figure 5.10: Logging times (wall clock) with regions of varying sizes for some PARSEC benchmarks ('native' input).

main thread in each program which put us in the region where all threads are active. The regions chosen were not actually buggy but if they were, we can get an idea of the time to log them with PinPlay logger. For logging time evaluation, we specified regions using a *skip* and *length* for the main thread. The evaluations were done on a pool of machines with 16 Intel Xeon ("Sandy Bridge E") processors (hyper-threading OFF) and 128GB of physical memory running SUSE Linux Enterprise Server 10.

Figure 5.10 presents the real/wall-clock time for logging regions of varying *length* values in the main thread. We only show the results for 5 "apps" and 3 "kernels" from the PASEC benchmark suite. For each benchmark we show the logging (with bzip2 pinball compression) time in seconds for regions of length 10 million to 1 billion dynamic instructions in the main thread. The total instructions in the region from all threads were 3-4 times more than the *length* in the main thread. The times shown do not include the time to fast-forward (using *skip*) to the region but just the time reported by the PinPlay logger between the start and the end of each region. Since the logger

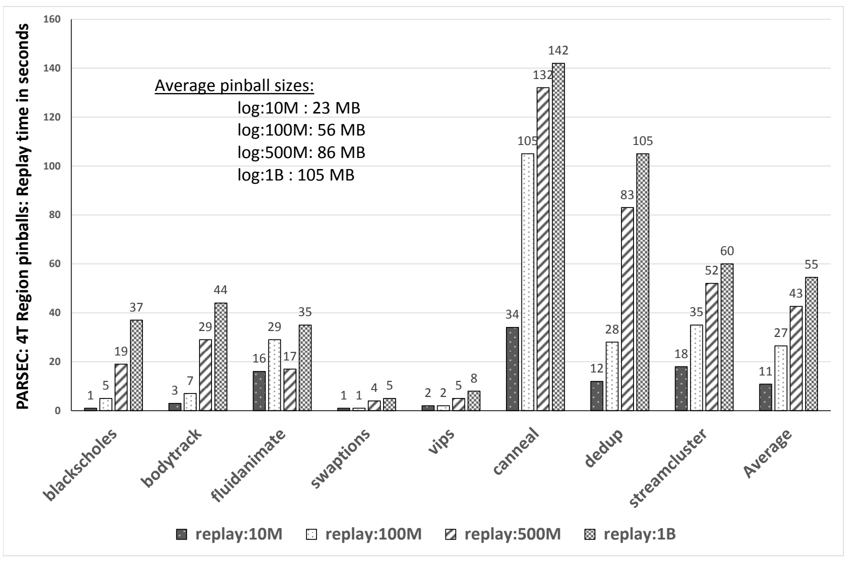


Figure 5.11: Replay times (wall clock) with pinballs for regions of varying sizes for some PARSEC benchmarks ('native' input).

does only minimal instrumentation before the region, the fast-forwarding can proceed at Pin-only speed. Figure 5.11 shows the time to replay the pinballs generated.

Both logging and replay times take from a few seconds (length 10 million) to a couple of minutes (length 1 billion). The actual times users see in practice depend on the length of the buggy region for their specific bug. In a study of 13 open source buggy programs reported in [77] the buggy region length (called *Window size* in the paper) was less than 10 million instructions for most programs with maximum being 18 million. We showed similar analysis result for some real buggy programs earlier in this section.

As described in [84], logging is more expensive than replay. However, logging will typically be done only once for capturing the buggy region. Replay will be done multiple times for cyclic debugging but since that is interlaced with user interaction and periods of user inactivity/thinking breaks, the replay overhead will be hardly noticeable (at least in our experience).

Finally, note that the pinballs (in tens to hundreds of MB in size) are small enough to be portable, so a buggy pinball can be transferred from one developer to another or from a customer site to a vendor site. The pinball size is *not* directly a function of region length but depends on memory access pattern and amount of thread interaction [84]. Hence pinballs for regions with length more than 1 billion instructions need not be substantially larger. In fact, sizes of pinballs for the entire execution of the five programs are in the range 4MB–145MB, much smaller than most region pinball sizes.

5.6.3 Slicing Overhead and Precision

There are two components to the slicing overhead: the time to collect dynamic information needed for efficient and precise dynamic slicing and the time to actually perform slicing. For the 8 PARSEC programs we tested, the average dynamic information tracing time for region pinballs with 1 million instructions (main thread) was 51 seconds. Once collected, the dynamic information can be used for multiple slicing sessions as PinPlay guarantees repeatability. For evaluating slicing time we computed slices for the last 10 read instructions (spread across five threads) for each region pinball. For the regions with 1 million instructions (main thread), the average size of the slice found was 218 thousand instructions and the average slicing time was 585 seconds.

We also measured the reduction in dynamic slice sizes achieved by pruning spurious dependences by identifying save/restore pairs. As the sizes of 10 dynamic slices for the 8 PARSEC programs are only slightly influenced with the spurious dependences prune, we omit the results here. Instead, we evaluated the effect of spurious dependences prune with five programs (ammp, apsi, galgel, mgrid, and wupwise) from SPECOMP 2001 benchmarks [4]. Figure 5.12 shows that, on average, dynamic slice sizes are reduced

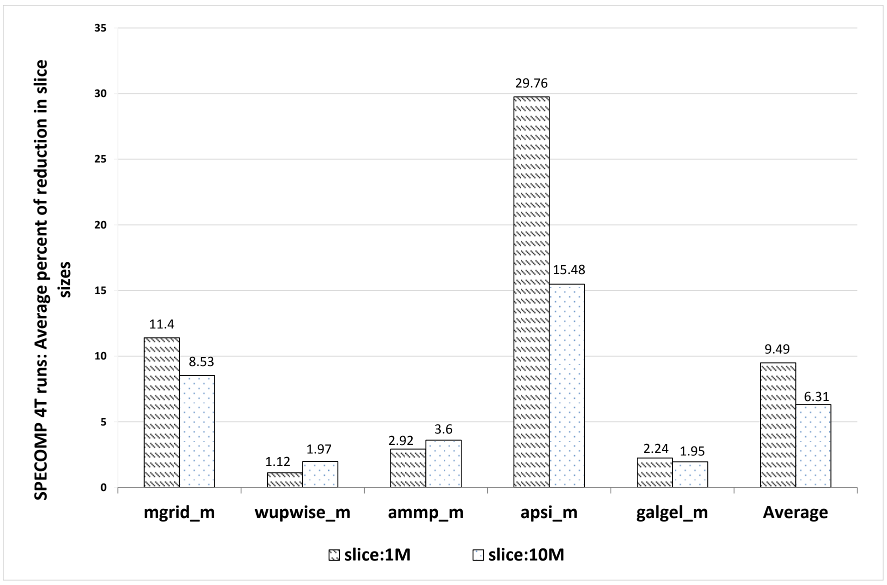


Figure 5.12: Removal of spurious dependences: average percentages of reduction in slice sizes over 10 slices for regions of length 1 million and 10 million dynamic instructions: SPECOMP (medium, test input).

by 9.49% (6.31%) for 1 million (10 million) instructions region pinballs ($MaxSave$ is set to 10 here).

5.6.4 Execution Slicing

When an execution slice is replayed using the slice pinball, the execution of code regions not included in the slice is skipped, making the replay faster. In Figure 5.13 we present the average replay time for 10 execution slice pinballs and the replay time for original, unsliced, pinball for regions of length 1 million instructions (main thread). Also included are average count of dynamic instructions in the slice pinballs as a percentage of total instructions in the full region pinball. As we can see, on average only 41% of dynamic instructions from a region pinball are included in an average slice. This makes the replay 36% faster on average. The results also show that the programmer will need to step through the execution of only 41% of executed instructions to localize the bug.

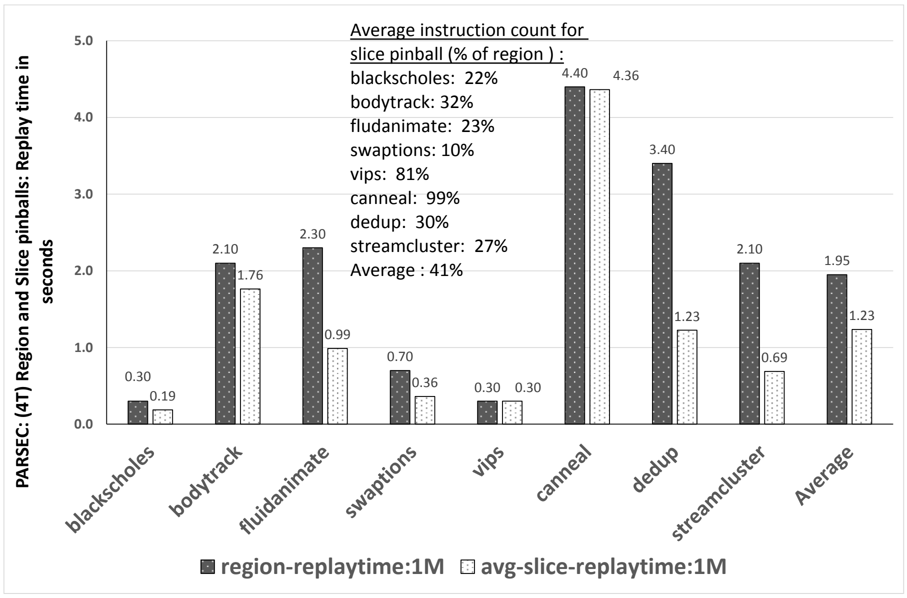


Figure 5.13: Execution slicing: average replay times (wall clock) for 10 slices for regions of length 1 million dynamic instructions: PARSEC ('native' input).

Thus cyclic debugging via slice pinball and execution slices greatly enhances debugging efficiency.

5.7 Summary

Cyclic debugging of multithreaded programs is challenging mainly due to run-to-run variation in program state. In this chapter, a set of program record/replay based tools was developed to address the above challenge. Through the development of `DrDebug`, it was demonstrated that the approach taken by `Qzdb` can be extended to multithreaded programs. While all capabilities of `Qzdb` were not included, `Drdebug` does extend substantial subset of new features of `Qzdb` to multithreaded programs. First, it provides tools for creating dynamic slices, checkpointing just the statements in a given dynamic slice, and navigating through the slice recording. By focusing on a buggy region instead of the entire execution and use of execution slices, `DrDebug` makes the time for recording, replaying, and dynamic slicing quite reasonable. Finally, like `Qzdb`, `DrDebug` works in conjunction with a real debugger (GDB) and with a graphical user interface front-end (KDbg).

Chapter 6

Related Work

The focus of this dissertation is on developing effective and efficient interactive debuggers which also accelerate fault localization. First, prior work on fault localization is presented, including work performed in context of interactive debuggers as well as stand alone fault localization tools and techniques. Second, prior work on improving efficiency via input and execution simplification are discussed and compared with the relevant input analysis based approach presented in this dissertation. Finally, prior work on runtime verification is summarized and contrasted with the bug specification based debugger generation presented in this dissertation.

6.1 Fault Localization

6.1.1 Techniques Employed by General Purpose Interactive Debuggers

GDB [26] provides low-level state alteration and inspection commands (e.g., breakpoint, print, and set) allowing the programmers to stop the program at points of interest and then inspect or alter the internal execution states. Chern et al. [13] improve breakpointing by allowing control-flow breakpoints. However, because such debuggers [26, 13] only provide low-level commands, and do not sufficiently guide the programmer in narrowing the source of error, even with their use the task of debugging remains very tedious. Extending the capabilities of such debuggers is also challenging.

Whyline [51] allows programmers to ask "why?" and "why not?" questions about program outputs and provides possible explanations based on program analysis, including static and dynamic slicing. In comparison to `Qzdb` and `DrDebug`, Whyline has several drawbacks. First, Whyline only supports postmortem analysis, while `Qzdb` and `DrDebug` support both backward navigation along dependence edges as well as forward single-stepping of the slice in a live debugging session. Second, since it is not integrated with a record/replay system, unlike `DrDebug`, Whyline does not support deterministic cyclic debugging. Third, programmers can only ask questions regarding program outputs, while with `Qzdb` and `DrDebug` programmers can compute a slice for any variable or register of interest.

Other debugging enhancements can be found in the following lines of work. Techniques have been developed to automatically generate breakpoints based upon automated fault location [33, 113]. The *vsdb* interactive debugger uses symbolic execution to display all possible symbolic execution paths from a program point [31]. Coca [21] allows programmers to query the execution trace. Gu et al. [29] propose a bug query language allowing programmers to fix their bugs by referring to similar resolved bugs. While these capabilities are complementary to ones presented in this dissertation, it should be noted that none of them take advantage of state alteration techniques.

6.1.2 Reverse Debugging Techniques

GDB also provides state rollback mechanism through the `checkpoint` and `restart` commands by employing the `fork` system call, but it does not provide such support for remote debugging. Moreover, its space overhead is prohibitively high due its use of the `fork`-based mechanism for checkpointing. In contrast, `Qzdb` uses incremental logging to reduce the space overhead.

UndoDB-gdb [99] does a much more efficient implementation of the record/replay and reverse debugging features. UndoDB "uses a 'snapshot-and-replay' technique, which stores periodic copy-on-write snapshots of the application and non-deterministic inputs"(quoted from [99]). TotalView debugger [98] has support for reverse debugging with ReplayEngine. It apparently works by forking multiple processes at different points in the recorded region and attaching to the right process on a 'step back' command.

The checkpoints in all the above debuggers are specific to a debug session and do not help with cyclic debugging. The real purpose of the reverse debugging commands is to find the points in the execution that affect a buggy outcome. Dynamic slicing is a more systematic way to find the same information that allows more focussed backward navigation. We believe reverse debugging can be supported in the `DrDebug` tool-chain by recording multiple pinballs and then replaying forward using the right pinball. Doing this using PinPlay's user-level checkpointing feature can be much more efficient than using operating system features (e.g., the fork mechanism).

VMWare supports replay debugging in their "Workstation" product between 2008 and 2011 [105]. Recording can be done either using a separate VMWare Workstation user-interface or with Microsoft's Visual Studio debugger. The recording overhead is extremely low because only truly non-reproducible events are captured by observing the operating system from a virtual machine monitor. However, the program is run on a single processor. That could make capturing certain multithreaded bugs hard. Also, the replay-based debugging works only with the virtual machine configuration where it was created. The record/replay framework `DrDebug` uses (PinPlay) does not require any special environment such as the virtual machine. Recording can be done in a program's native environment and then the recording can be replayed and debugged on any other machine.

6.1.3 State Alteration Techniques

Many state alteration techniques have been developed in the context of automated fault location [116, 43, 10], in contrast to their integration in interactive debuggers as carried out in this dissertation. Zhang et al. [116] proposed predicate switching to constrain the search space of program state changes explored during bug location. Corrupted memory location suppression [43] attempts to identify the root cause of memory bugs by iteratively suppressing the potential cause of the memory failure. The statement involved in the final suppression is then highly likely to be the root cause of the memory bug. In [44], execution suppression is extended to identify the root cause for concurrency bugs. Jeffrey et al. [42] proposed value replacement to automatically pinpoint erroneous code. Chandra et al. [10] reports repair candidates based on value replacement. Gu et al. [29] proposed a bug query language allowing programmers to fix their bugs by referring to similar resolved bugs. In contrast, `Qzdb` generalizes the state alteration based automatic fault localization techniques (i.e., predicate switching and execution suppression) by introducing them into a general-purpose debugger.

6.1.4 Static and Dynamic Slicing Techniques

Static and dynamic slicing [97, 104, 54, 73, 51, 52, 53, 119, 120, 106, 122, 1, 12, 30, 32] have been widely recognized as being helpful for debugging. *Static slicing* [97, 104] calculates all the program statements that may influence the value of the specified variable at a particular program point, directly and indirectly, via static control and data dependences. Static slicing has been extended for concurrent programs in [54, 73]. *Dynamic slicing* [52, 53, 97, 122] calculates all the program statements that actually affect the value of the specified variable at a particular program point during a specific dynamic execution via dynamic control and data dependence. Dynamic slices are a subset of static slices. A series of efforts have been aimed at improving the effectiveness and efficiency of dynamic slicing [120, 106, 119, 1, 12, 30, 32].

Thin Slicing [93] only considers data dependences which helps compute and copy a value to the specified variable. Tallam et al. extended dynamic slicing to detect data races [94]. Weeratunge et al. [102] presented dual slicing by leveraging both passing and failing runs. However, unlike `DrDebug`, neither approach is designed to be integrated with a record/replay system. Moreover, the dynamic slicing algorithm used by `DrDebug` is highly precise via its use of CFG refinement via dynamic jump targets and bypassing of spurious data dependence resulting from save/restore pairs. Slice pruning [118] eliminates unnecessary dependences according to programmers' feedback. However, the time overhead is unacceptable in an interactive debugging scenario because of the time-consuming value profiling and the costly calculation of alternate set and confidence for each statement. `Qzdb` generalizes the idea of slice pruning to only exclude dependence edges from the generalized slice that are related to user-specified variables.

6.1.5 Techniques for Locating Memory-Related Errors

Purify [35] and Valgrind [79] detect the presence of memory bugs via dynamic binary instrumentation. CCured [78] uses type inference to classify pointers and applies dynamic checks according to the classification for memory safety. Rx [87] recovers from a crash by rolling back the execution and reexecuting after changing the execution environment. AccMon [124] detects memory-related bugs by capturing violations of program counter based invariants. DieHard tolerates memory errors through randomized memory allocation and replication [7]. Exterminator dynamically generates runtime patches based upon runtime information [82]. Nagarakatte et al. use compile-time transformations for ensuring spatial [71] and temporal safety [72] of programs written in C. Bond et al.'s approach [9] tracks the origins of unusable values; however, it can only track the origin of Null and undefined values while our VPCs capture not only origin, but propagation for any specified variable.

MemTracker [100] provides a unified architectural support for low-overhead programmable tracking to meet the needs for different kinds of bugs. FindBugs [36] leverages bug patterns to locate bugs. Algorithmic (or declarative) debugging [91] is an interactive technique where the user is asked at each step whether the prior computation step was correct [92]. Program synthesis has been used in prior work to automatically generate programs from specifications at various levels: types [65], predicates or assertions/goals [66]; however no prior work on synthesis has investigated specification at the operational semantics level in the context of debugging.

6.1.6 Statistical Debugging Techniques

Renieris and Reiss [90] identify faulty code by considering differences in statements executed by passing and failing runs. Tarantula [46, 47] prioritizes statements based on their appearance frequency in failing runs versus passing runs. Liblit et. al. presented *Cooperative Bug Isolation* [57, 58, 59] that locates bug root causes based on statistical analysis of program executions. Liu et. al. presented *SOBER* [60] which models evaluation patterns of predicates in both passing and failing program runs respectively and identifies a predicate as bug-related if its evaluation pattern in failing runs differs significantly from that in passing ones. In general, these approaches rely on a large test suite (including enough passing and failing runs) that may not be available in practice.

6.2 Techniques for Input and Execution Simplification

6.2.1 Prior Forms of Relevant Input Analysis

Prior relevant input analyses such as lineage tracing [114, 14, 6] identify the subset of inputs that contribute to a specified output by considering data dependence only [114], both data dependence and control dependence [14], or both data dependence and strict control dependence [6]. However, none of these approaches differentiate the *role* and *strength* inputs play in computing a specified value. The relevant input analysis presented in this dissertation characterizes the *role* and *strength* of inputs play in the computation of different values during a program execution. Moreover, we are the first to leverage the result of relevant input analysis to speed up the delta debugging algorithm which can greatly simplify failing input as well as the program execution.

6.2.2 Input Reduction via Delta Debugging

Given a program input on which the execution of a program fails, delta debugging [112] automatically simplifies the input such that the resulting simplified input causes the same failure. This technique can then be applied to passing and failing executions to automatically identify cause-effect chains [111].

Hierarchical Delta Debugging [70] leverages the structure inside program inputs to accelerate the search for minimal input. The hierarchical input decomposition technique presented in this dissertation has several advantages in comparison to hierarchical delta debugging (HDD) [70]. First, HDD requires that the initial failure-inducing input be well-formed; otherwise, the parser which HDD is based on will fail. Note that HDD only generates syntactically valid input. However, it is common that programs often fail because of ill-formed input. Second, HDD users must provide infrastructure for input parsing, unparsing a configuration, and pruning nodes from the input tree for different languages, which turns out to be non-trivial [70].

6.2.3 Execution Reduction Techniques

Several existing works on execution reduction [123, 95, 56, 40] either reduce the tracing overhead during replay [123, 95] or replay overhead [56, 40]. In [123] and [95] authors support tracing and slicing of long-running multi-threaded programs. They leverage meta slicing to keep events that are in the transitive closure of data and control dependences with respect to the event specified as slicing criterion. However, the slicing criterion can only be events (e.g., I/O) captured during the logging phase. On the other hand, `DrDebug` can reduce the replay pinball for any variable. Lee et al. [56] proposed a technique to record extra information during logging and then leveraged it to reduce the replay log in unit granularity based on programmers' annotation of unit. `DrDebug`'s execution region enables programmers to only log and fast forward to the reasonable small buggy region during replay. Thus, `DrDebug` can reduce the region pinball to a slice pinball at finer granularity without requiring unit annotations by the programmer. LEAN [40] presents an approach to remove redundant threads with delta debugging and redundant instructions with dynamic slicing while maintaining the reproducibility of concurrency bugs. However, the overhead of delta debugging can be very high as it requires repeated execution of replay runs. More importantly, none of these works support stepping through a slice in a live debugging session.

6.3 Runtime Verification Techniques

Monitor-oriented programming (MOP) [69] and Time Rover [96] allow correctness properties to be specified formally (e.g., in LTL, MTL, FSM, or CFG); code generation is then used to yield runtime monitors from the specification. Monitor-oriented programming (MOP) [11, 69] combines formal specification with runtime monitoring. In MOP, correctness properties can be specified in LTL, as a FSM, or as a CFG. Then, from a specification, a low-overhead runtime monitor is synthesized to run in AspectJ (i.e., use aspect-oriented programming [49] in JavaMOP [11]) or on the PCI bus (in BusMOP [85]) to monitor the program execution and detect violations of the specification. Time Rover [20, 96] combines LTL, MTL and UML specification with code generation to yield runtime monitors for formal specifications.

PQL [68] and PTQL [27] allow programmers to query the program execution history, while tracematches [2] allows free variables in trace matching on top of AspectJ. GC assertions [89] allow programmers to query the garbage collector about the heap structure. Jinn [55] synthesizes bug detectors from state machine for detecting foreign function interface.

Ellison and Roşu [22] define a general-purpose semantics for C with applications including debugging and runtime verification; in our semantics we only expose those reduction rules that help specify memory debuggers, but our approach works for the entire x86 instruction set and sizable real-world programs including library code.

Compared to all these approaches, the work on bug specification presented in this dissertation differs in several ways. The prior approaches are adept at specifying properties and generating runtime checkers (which detect what property has been violated). In contrast the approach presented in this dissertation points out *where*,*why*, and *how* a property is violated; it also introduces value propagation chains to significantly reduce the effort associated with bug finding and fixing.

Chapter 7

Conclusions and Future Directions

7.1 Contributions of this Dissertation

The main contributions of this dissertation are in making interactive debuggers effective, extensible, and efficient for singlethreaded and multithreaded programs via the design and prototyping of the `Qzdb` and `DrDebug` debuggers. The innovations in interactive debugging are made possible by a series of novel dynamic analysis techniques developed in this dissertation. In conclusion, the following research questions are addressed in this dissertation.

7.1.1 Can the effectiveness of general-purpose debuggers be improved using dynamic analysis techniques?

Existing general-purpose debuggers [26, 13] only provide low-level commands, and do not effectively guide the programmer in narrowing the root cause of bugs, thus it is very tedious to debug practical programs. Through the development of the `Qzdb` debugger, this dissertation shows that an interactive debugger built around a powerful dynamic analysis framework can overcome the above drawbacks. `Qzdb` supports commands that allow the programmer to narrow down the bugs' location successively to smaller and smaller code regions based upon high-level commands supported via sophisticated dynamic analysis techniques. `Qzdb` supports state alteration commands to affect the control flow and suppress the execution of statements/functions, allowing the programmer to quickly narrow faulty code down to a function. The state inspection techniques allow efficient examination of large code regions by navigating and pruning dynamic slices and zooming in on chains of dependences. Finally, the programmer can zoom to a small set of statements in a slice by single stepping at those statements and examining program state. The debugger is extensible as a new command can be added by simply extending the user interface and implementing the dynamic analysis needed to implement the new command. Case studies based on real reported bugs demonstrate `Qzdb` can greatly speed up bug understanding and fixing.

7.1.2 Can an interactive debugger support systematic extensibility to allow incorporation of specialized bug detection algorithms?

Since the detection of specific kinds of bugs (e.g., memory-related bugs) is tedious using general-purpose debuggers [26, 13], programmers use tools tailored to specific kinds of bugs (e.g., buffer overflows [71, 18], dangling pointer dereferences [19], and memory leaks [108, 83]). However, to use the appropriate tool the programmer needs to first know what kind of bug is present in the program. Second, even given the bug report from specialized bug detector, programmers may still need to resort to general-purpose debuggers to understand and finally fix the bug (e.g., why is a dereferenced pointer NULL and how did the NULL value propagates to the failure point?). Finally, inclusion of such capabilities in existing general-purpose debuggers is also a daunting task.

This dissertation addresses the above challenge by presenting a novel approach for constructing debuggers automatically from bug specifications for memory-related bugs. With the presented bug specification language, programmers can declaratively specify bugs and their root causes, using just 1 to 4 predicates for all the 6 bugs types studied. The automated translation was used to generate an actual debugger that works for arbitrary C programs running on the x86 platform. The bug detection has been proven to be sound with respect to a low-level operational semantics, i.e., bug detectors fire prior to the machine entering an error state.

In addition to bug detection rules, programmers can also specify *locator rules* that rely upon the novel concept of value propagation chain. Value propagation chains capture data dependences introduced by propagating (copying) existing values—a small subset of dynamic slices. For each bug kind, the value propagation chain specifies how the value involved in the bug manifestation relates to the bug's root cause. These chains drastically simplify the process of detecting and locating the root cause of memory bugs. For the real-world programs we have studied, programmers have to examine just 1 to 16 instructions.

7.1.3 Can a debugger built upon powerful dynamic analysis techniques be made efficient enough to handle real-world bugs?

As expensive dynamic analysis techniques are needed for both high-level commands (i.e., predicate switching, dynamic slicing) and automatically generated debuggers (i.e., generated double free bug detector and locator), the overhead of `Qzdb` can be high. This is particularly true for long program executions. This dissertation alleviates this problem by simplifying a failing program input as well as its dynamic execution with the guarantee that the same failure manifests in the simplified execution. Then programmers begin the debugging task with the simplified execution instead of the original long execution. This dissertation again achieves input and execution simplification by presenting a novel dynamic analysis, named *relevant input analysis*, and uses its result to enhance the *delta debugging* algorithm.

The *relevant input analysis* characterizes the *role* and *strength* of inputs in the computation of different values during a program execution. The *role* indicates whether a computed value is *derived* from an input value or its computation is simply *influenced* by an input value. The *strength* indicates if role relied upon the precise value of the input or it is among one of many values that can play a similar role. The relevant input analysis is then used to prune, as well as guide, and hence accelerate, the *delta debugging* algorithm. The latter automatically simplifies the input such that the resulting simplified input (specifically, 1-minimal input, i.e., an input from which removal of any entity causes the failure to disappear) causes the same failure. Experiments show that *relevant input analysis* based input simplification algorithm is both efficient and effective – it only requires 11% to 21% of test runs needed by the standard delta debugging algorithm and generates even smaller inputs.

7.1.4 Can the developed approach for interactive debugging be extended to handle multithreaded programs?

Debugging multithreaded programs poses additional challenges: it may take a very long time to fast-forward to the beginning of a buggy region and program states (the outcome of system calls, thread schedule, shared memory access order, etc.) vary from run to run. To address these challenges, this dissertation presents `DrDebug`, allowing deterministic and efficient cyclic debugging based upon PinPlay, a record and replay framework. Deterministic and efficient cyclic debugging with `DrDebug` is achieved by repeated replay of buggy *execution region* or an *execution slice* of a buggy region.

With the help of *execution region*, only the buggy regions get replayed during each debugging session, avoiding repeated and time-consuming fast-forwarding to the begin of buggy region. In addition, *execution slices* can improve the debugging efficiency even further by skipping bug-irrelevant program execution (i.e., statements not in the slice of the failure point) during replay. `DrDebug` also allows the user to step from the execution of one statement in the slice to the next while examining the values of variables in a live debugging session. Case studies of real reported concurrency bugs show that the buggy *execution region* size is less than 1 million instructions and the lengths of buggy *execution region* and *execution slice* are less than 15% and 7% of the total execution respectively.

7.2 Future Directions

While the capabilities and efficiency of `Qzdb` and `DrDebug` could be further enhanced, a major direction for future work is the application of the principles we described to create a debugger for applications developed for mobile platforms.

With the fast-growing and highly-competitive mobile application ecosystems, user experience is crucial for the survival of a mobile application ("app"). Thus, it is important to provide effective and efficient debugging support for software developed for mobile apps. However, prior software research on mobile systems has mainly focused on security [3, 23, 24], testing [5, 63], performance [88, 103], and specialized bug detection tools (e.g., data race detection [37, 64]), and only a few attempts have been made to provide general-purpose debugging support [28, 34]. Therefore, it would be interesting and beneficial to extend the techniques we have developed in this dissertation to leading smartphone platforms such as Android [41], by building a dynamic analysis framework, and then developing effective debugging support such as dynamic slicing and relevant input analysis.

7.2.1 Challenges for Android Platform

The differences between mobile apps and desktop (or server) programs introduce many debugging challenges, as described below:

- Android apps are largely written in Java and compiled to DEX bytecode, which is executed by the Dalvik VM interpreter. Thus, a dynamic analysis framework for the Android platform needs to be implemented, e.g., via Dalvik VM instrumentation (monitoring and instrumenting app execution when the Dalvik VM interprets the bytecode).
- Android apps can invoke native libraries (provided by the Android Framework or third-party) written in C/C++, and can be executed directly without the need of interpretation by Dalvik VM. Therefore, the execution of native libraries cannot be monitored by a Dalvik VM instrumentation based dynamic analysis framework. For accurate dynamic analysis, a native code (binary) instrumentation framework is required (e.g., Pin).
- Android implements JIT compilation for most apps. Similar to native libraries, such JIT code execution escapes the dynamic monitoring by a Dalvik VM-based dynamic analysis framework, so JIT compilation must either be turned off (as previous work [37]) or tracked differently via a native code (binary) instrumentation framework (e.g., Pin).

- The Dalvik VM optimizes the bytecode during running (e.g., JIT), so static analysis (e.g., post-dominator analysis needed for dynamic control dependence detection) on app bytecode might not always be accurate.
- The IPC message passing system of Android allows apps to share messages through intents; data can be shared via "Bundles." Thus, the dynamic analysis framework needs to capture external influence due to IPC (e.g., to correctly identify inter-apps data/value/address dependence caused by IPC).
- Android apps have much richer types of user inputs (complex gestures, GPS, other physical sensors) than pc/server programs. The dynamic analysis framework should be able to capture those rich inputs.
- Some inputs (e.g., complex gestures) have strict timing constraints, and changing the timing between events may yield wrongly-identified gestures, finally wrong behavior. Such strict timing constraints impose tremendous challenges for the dynamic analysis framework (which inevitably incurs high runtime overhead). Thus, the instrumented/monitored runs may violate the original timing constraints, and very likely interfere with app execution.
- Mobile platforms have limited memory/disk resources. The dynamic analysis framework needs to cope with such resource limitations.
- Finally, Google has introduced a new runtime, ART, from Android 4.4, which actually compiles the app bytecode into native binary during installation. With ART, a dynamic analysis framework based on a binary instrumentation framework, e.g., Pin is preferable. With ART, both app bytecode and Android framework/third party native libraries are compiled to native code and executed directly; thus the dynamic analysis framework based on a binary instrumentation framework can capture the entire execution.

7.2.2 Dynamic Analysis Framework for Android Apps

The dynamic analysis framework should support efficient def-use tracing and online control dependence detection, which are the core components of many dynamic analysis techniques (e.g., dynamic slicing, relevant input analysis, and information flow tracking).

In the presence of inter-process communication (which is widely used between different Android apps), the dynamic analysis can be conducted in two different ways:

- *system tracking*: tracking all related apps as well as their interactions via IPC. By adding dependence edges between the sender/receiver of IPC, we can build a system-wide dependence graph, and thus conduct system-wide dynamic analysis. This option incurs higher runtime and space overhead, while providing more accurate dynamic analysis.
- *process tracking*: only tracking the app of interest, and then capturing the system as well as all other apps' external influence manually (e.g., core system components have given information flow semantics) or with the help of users' specification (i.e., the system and all other apps can be treated as a black box). This option is more runtime efficient but sacrifices accuracy.

Dependence Tracking for Dynamic Slicing. The execution of bytecode, as interpreted by the Dalvik VM, can be monitored to gather the def-use trace and control dependence information. Due to the limited memory and space resources on mobile devices, all results (including per-thread def-use trace, shared memory access ordering, and detected control dependence) should be saved on a pc/server for storage and processing.

As Android apps are multi-threaded, there is a need to track the shared memory access ordering which can be captured via a software implementation of a hardware approach [81]. Shared memory access ordering is needed to construct the global def-use trace. The *logcat* utility can be modified to dump the per-thread def-use trace. Intermediate post dominator for each basic block is needed for online control dependence detection. Redexer [45], an OCaml-based DEX code rewriter, can be used to create the initial post dominator tree, and the intermediate post dominator for each basic block is computed based on the post dominator tree. The online control dependence detection algorithm proposed by Xin et al. [106] can be implemented by modifying the Dalvik VM.

In addition to the bytecode, native code must also be monitored and analyzed. There can be four kinds of native code: system libraries, third-party libraries, internal VM methods, and JIT compilation. The JIT compilation can be disabled to handle the fourth case. However, for system and third-party libraries and internal VM methods, an instrumentation framework for native code (e.g., Pin) is needed to allow accurate def-use tracing and control dependence detection.

Integration with a Record/Replay System. As both def-use tracing and online control dependence detection require heavy-weight instrumentation, they will incur significant perceivable delays to the user. Even worse, Android apps inputs (e.g., complex gestures) have strict timing constraints, and changing the timing between events (because of the execution of instrumented analysis) may yield wrongly identified gestures. Thus, the dynamic analysis framework must be integrated with a record/replay system such as RERAN [28]. RERAN satisfies two requirements: (a) imperceivable (or tolerable) recording delay; and (b) ensuring the same gestures are identified during record/injected during replay. With the integration of such a record/replay system, the def-use tracing and online control dependence detection can be performed during the replay phase. The users can interact with the apps only during recording phase (with negligible delays), and then the tracked event sequences are injected during replay phase, during which the def-use tracing and online control dependence detection can be performed.

Relevant Input Analysis. Compared to inputs for desktop or server programs, mobile app inputs are much richer and more challenging to capture. Rules can be developed to capture different external sensor inputs (GPS, accelerometer, camera, etc.), and then save the captured inputs to a pc/server.

Due to memory resource constraints, at this point it is infeasible to conduct online relevant input analysis on mobile devices. Once the program inputs, def-use trace and control dependences are saved on a pc/server, a forward traversal of the trace can be carried out to build the value dependence, address dependence and control dependence, and then roles and strength of inputs are characterized based on the dependence.

The relevant input analysis can be very useful for program comprehension, and its result has many other applications:

- *Delta Debugging:* simplifying failure-inducing inputs (e.g., gestures) can identify which user gestures triggered a bug. The relevant input analysis can accelerate delta debugging (e.g., 1-minimal input searching).
- *Test Case Generation:* generating inputs covering different branches/paths based on existing inputs (e.g., gestures).
- *Security:* data dependences may be obfuscated as control dependences to avoid detection. The relevant input analysis helps capture obfuscated vulnerabilities.

In summary, the principles and capabilities developed in this dissertation would be useful in the context of a mobile platform like Android; however, many challenges must be overcome to develop a debugging tool for Android.

Bibliography

[1] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. *Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering*, pages 143–151, October 1995.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 345–364, 2005.

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

[4] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In *In Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.

[5] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 641–660, 2013.

[6] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 13–24, 2010.

[7] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, June 2006.

[8] C. Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[9] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *Proceedings*

of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, OOPSLA '07, pages 405–422, 2007.

[10] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceeding of the 33rd international conference on Software engineering*, pages 121–130. ACM, 2011.

[11] F. Chen and G. Roşu. Mop: An efficient and generic runtime verification framework. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 569–588, 2007.

[12] T. Y. Chen and Y. Y. Cheung. Dynamic program dicing. *Proceedings of the IEEE International Conference on Software Maintenance*, pages 378–385, September 1993.

[13] R. Chern and K. De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 96–106, 2007.

[14] J. Clause and A. Orso. Penumbra: automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of Symposium on Software Testing and Analysis*, pages 249–260, 2009.

[15] H. Cleve and A. Zeller. Locating causes of program failures. *27th International Conference on Software Engineering*, pages 342–351, May 2005.

[16] B. De Bus, B. De Sutter, L. Van Put, D. Chanet, and K. De Bosschere. Link-time optimization of arm binaries. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'04)*, pages 211–220, 7 2004.

[17] Ibm tutorial on debugging–debugging using comments. https://www-927.ibm.com/ibm/cas/hspc/Resources/ Tutorials/debug_2.shtml.

[18] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 162–171, 2006.

[19] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 269–280, 2006.

[20] D. Drusinsky. The temporal rover and the atg rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, 2000.

[21] M. Ducassé. Coca: an automated debugger for c. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 504–513, 1999.

[22] C. Ellison and G. Rosu. An executable formal semantics of c with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 533–544, 2012.

[23] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, 2010.

[24] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[25] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[26] Gdb page. http://www.gnu.org/software/gdb/.

[27] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 385–402, 2005.

[28] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 72–81, 2013.

[29] Z. Gu, E. Barr, and Z. Su. Bql: capturing and reusing debugging knowledge. In *Proceeding of the 33rd international conference on Software engineering*, pages 1001–1003. ACM, 2011.

[30] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. *IEEE/ACM International Conference on Automated Software Engineering*, pages 263–272, November 2005.

[31] R. Hähnle, M. Baum, R. Bubel, and M. Rothe. A visual interactive debugger based on symbolic execution. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 143–146, 2010.

[32] C. Hammer, M. Grimme, and J. Krinke. Dynamic path conditions in dependence graphs. *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 58–67, January 2006.

[33] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei. Vida: Visual interactive debugging. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 583–586, 2009.

[34] S. Hao, D. Li, W. G. Halfond, and R. Govindan. Sif: A selective instrumentation framework for mobile applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 167–180, 2013.

[35] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. *Proceedings of the USENIX Winter Technical Conference*, pages 125–136, 1992.

[36] D. Hovemeyer and W. Pugh. Finding bugs is easy. pages 132–136, 2004.

[37] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. 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.

[38] `http://flex.sourceforge.net/`. Flex homepage.

[39] `http://www.gnu.org/software/bison/`. Bison homepage.

[40] J. Huang and C. Zhang. Lean: simplifying concurrency bug reproduction via replay-supported execution reduction. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12, pages 451–466, 2012.

[41] IDC. Android pushes past 80% market share while windows phone shipments leap 156.0% year over year in the third quarter, Novemeber 2013.

[42] D. Jeffrey, N. Gupta, and R. Gupta. Fault localization using value replacement. *International Symposium on Software Testing and Analysis*, pages 167–178, July 2008.

[43] D. Jeffrey, N. Gupta, and R. Gupta. Identifying the root causes of memory bugs using corrupted memory location suppression. *IEEE International Conference on Software Maintenance*, pages 356–365, September 2008.

[44] D. Jeffrey, Y. Wang, C. Tian, and R. Gupta. Isolating bugs in multithreaded programs using execution suppression. *Software: Practice and Experience*, 41(11):1259–1288, 2011.

[45] Jinseong Jeon and Kristopher Micinski and Jeffrey S. Foster. Redexer, September 2013. `http://www.cs.umd.edu/projects/PL/redexer/index.html`.

[46] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[47] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, May 2002.

[48] Kdbg page. http://www.kdbg.org/.

[49] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97*, pages 220–242, 1997.

[50] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, 2005.

[51] A. J. Ko and B. A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, 2008.

[52] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[53] B. Korel and J. Rilling. Application of dynamic slicing in program debugging. *Proceedings of the International Symposium on Automated Analysis-Driven Debugging*, pages 43–58, May 1997.

[54] J. Krinke. Context-sensitive slicing of concurrent programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 178–187, 2003.

[55] B. Lee, B. Wiedermann, M. Hirzel, R. Grimm, and K. S. McKinley. Jinn: Synthesizing dynamic bug detectors for foreign language interfaces. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 36–49, 2010.

[56] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang. Toward generating reducible replay logs. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI'11, pages 246–257, 2011.

[57] B. Liblit. Cooperative bug isolation. *Ph.D. Thesis, The University of California, Berkeley*, 2004.

[58] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 141–154, June 2003.

[59] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 15–26, June 2005.

[60] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. SOBER: Statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, September 2005.

[61] G. Lueck, H. Patil, and C. Pereira. Pinadx: an interface for customizable debugging with dynamic instrumentation. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 114–123, 2012.

[62] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. 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.

[63] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, 2013.

[64] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, 2014.

[65] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference*

on Programming Language Design and Implementation, PLDI '05, pages 48–61, 2005.

[66] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, Jan. 1980.

[67] Maple sources. https://github.com/jieyu/maple.

[68] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using pql: A program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 365–383, 2005.

[69] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu. An overview of the MOP mop-jsttt11 framework. *International Journal on Software Techniques for Technology Transfer*, pages 249–289, 2011.

[70] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE'06, pages 142–151, 2006.

[71] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 245–258, 2009.

[72] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, 2010.

[73] M. G. Nanda and S. Ramesh. Interprocedural slicing of multithreaded programs with applications to java. *ACM Trans. Program. Lang. Syst.*, 28(6):1088–1144, Nov. 2006.

[74] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 216–227, 2006.

[75] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems(SIGMETRICS)*, SIGMETRICS'06, pages 216–227, 2006.

[76] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. pages 284–295, 2005.

[77] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd annual international symposium on Computer Architecture(ISCA)*, ISCA'05, pages 284–295, 2005.

[78] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. *Symposium on Principles of Programming Languages*, pages 128–139, January 2002.

[79] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 89–100, June 2007.

[80] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. pages 89–100, 2007.

[81] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[82] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. pages 1–11, 2007.

[83] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 397–407, 2009.

[84] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: A framework for deterministic replay and reproducible analysis of parallel programs. In *CGO '10: Proceedings of the 2010 International Symposium on Code Generation and Optimization*, CGO'10, pages 2–11, 2010.

[85] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *Proceedings of the 29th IEEE Real-Time System Symposium (RTSS'08)*, RTSS'08, pages 481–491, 2008.

[86] Program record/replay (PinPlay) toolkit. http://www.pinplay.org.

[87] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies – a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3):Article 7 (1–33), August 2007.

[88] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 107–120, 2012.

[89] C. Reichenbach, N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What can the gc compute efficiently?: a language for heap assertions at gc time. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 256–269, 2010.

[90] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 30–39, October 2003.

[91] E. Y. Shapiro. *Algorithmic Program DeBugging.* MIT Press, Cambridge, MA, USA, 1983.

[92] J. Silva. A survey on algorithmic debugging strategies. *Adv. Eng. Softw.*, 42(11):976–991, Nov. 2011.

[93] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 112–122, 2007.

[94] S. Tallam, C. Tian, and R. Gupta. Dynamic slicing of multithreaded programs for race detection. In *ICSM'08*, pages 97–106, 2008.

[95] S. Tallam, C. Tian, R. Gupta, and X. Zhang. Enabling tracing of long-running multithreaded programs via dynamic execution reduction. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 207–218, 2007.

[96] Time rover homepage. http://www.time-rover.com.

[97] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[98] Reverse debugging with replayengine. http://www.roguewave.com/ products/totalview/replayengine.aspx.

[99] Undodb page. http://undo-software.com/product/undodb-man-page.

[100] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Memtracker: An accelerator for memory debugging and monitoring. *ACM Trans. Archit. Code Optim.*, 6(2):5:1–5:33, July 2009.

[101] Visual studio debugger–edit and continue. http://msdn.microsoft.com/en-us/library/bcew296c.aspx.

[102] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA'10, pages 253–264, 2010.

[103] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 137–148, 2012.

[104] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[105] Vmware's replay debugging offering. http://www.replaydebugging.com/.

[106] B. Xin and X. Zhang. Efficient online detection of dynamic control dependence. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 185–195, 2007.

[107] B. Xin and X. Zhang. Memory slicing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA'09, pages 165–176, 2009.

[108] G. Xu, M. D. Bond, F. Qin, and A. Rountev. Leakchaser: Helping programmers narrow down causes of memory leaks. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 270–282, 2011.

[109] J. Yu and S. Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA'09, pages 325–336, 2009.

[110] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA '12: Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'12, pages 485–502, 2012.

[111] A. Zeller. Isolating cause-effect chains from computer programs. *10th International Symposium on the Foundations of Software Engineering*, pages 1–10, November 2002.

[112] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[113] C. Zhang, D. Yan, J. Zhao, Y. Chen, and S. Yang. Bpgen: An automated breakpoint generator for debugging. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 271–274, 2010.

[114] M. Zhang, X. Zhang, X. Zhang, and S. Prabhakar. Tracing lineage beyond relational operators. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 1116–1127, 2007.

[115] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting severe concurrency bugs through an effect-oriented approach. *ACM SIGPLAN Notices*, 45(3):179–192, March 2010.

[116] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. *Proceedings of the 28th International Conference on Software Engineering*, pages 272–281, May 2006.

[117] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[118] X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.

[119] X. Zhang and R. Gupta. Cost effective dynamic program slicing. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 94–106, June 2004.

[120] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *IEEE/ACM International Conference on Software Engineering*, pages 319–329, May 2003.

[121] X. Zhang, R. Gupta, and Y. Zhang. Precise dynamic slicing algorithms. *IEEE/ACM International Conference on Software Engineering*, pages 319–329, May 2003.

[122] X. Zhang, H. He, N. Gupta, and R. Gupta. Experimental evaluation of using dynamic slices for fault location. *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging*, pages 33–42, September 2005.

[123] X. Zhang, S. Tallam, and R. Gupta. Dynamic slicing long running programs through execution fast forwarding. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 81–91, 2006.

[124] P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. P. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. *37th Annual International Symposium on Microarchitecture*, pages 269–280, December 2004.

Appendix A

Soundness Proof

We use Σ as a shorthand for a legal state $\langle H; F; \overline{S}; P; k; \sigma; f; e\rangle$, and *Error* as a shorthand for an error state. Hence, the condensed form of the reduction relation for legal transitions is $\Sigma \longrightarrow \Sigma'$, while transitions $\Sigma \longrightarrow Error$ represent bugs. We name state Σ as $\Sigma_e = \langle H_e; F_e; \overline{S}_e; P_e; k_e; \sigma_e; f_e; e\rangle$, and Σ' as $\Sigma'_e = \langle H'_e; F'_e; \overline{S}'_e; P'_e; k'_e; \sigma'_e; f'_e; e'\rangle$. If expression e generates an event, we name it as ν_e, that is, $\sigma'_e = \sigma_e, \nu_e$.

At a high level, our notion of soundness can be expressed as follows: if the abstract machine, in state Σ, would enter an error state next, which is the "ground truth" for a bug, then the user-specified bug detectors, defined in terms of just e and σ, must fire.

The proof of soundness relies on several key definitions and lemmas. We first define well-formed states, then prove that reductions to non-error states preserve well-formedness, and finally the soundness theorem captures the fact that the premises of error transition rules in fact satisfy the user-defined bug specification, hence bugs will be detected. Well-formed states are defined as follows:

Definition A.0.1 (Well-formed states)**.** *A state* $\Sigma = \langle H; F; \overline{S}; P; k;\ \sigma; f; e\rangle$ *is well-*

formed if:

1. $H \cap F = \emptyset$

2. $(H \cup F) \cap (\bigcup\limits_{S \in \overline{S}} S) = \emptyset$

The first part says that block id's cannot simultaneously be in the heap H and in the freed set F, while the second part ensures that block id's cannot simultaneously be both in the heap H (or the freed set F) and in the stack $\overline{S}$.

Lemma A.0.2 (Empty initial state)**.** *Suppose the initial state, denoted* Σ_0*, is* $\langle H_0; F_0; \overline{S}_0; P_0; k_0; \sigma_0; f_0; a$
In this state the heap, stack, and freed set are empty, i.e.:

1. $H_0 = \emptyset$

2. $\overline{S}_0 = \emptyset$

3. $F_0 = \emptyset$

Proof. The lemma holds by construction. □

Lemma A.0.3 (Well-formed initial state)**.** Σ_0 *is well-formed.*

Proof. The lemma follows immediately from Definition A.0.1. □

Next, we introduce a lemma to prove that non-error transitions keep the state well-formed.

Lemma A.0.4 (Preservation of well-formedness)**.** *If* Σ *is well-formed and* $\Sigma \longrightarrow \Sigma'$*, and* Σ' *is not an error state, then* Σ' *is well-formed.*

The proof is by induction on the reduction $\Sigma \longrightarrow \Sigma'$. Intuitively, this lemma states that, since the state always stays well-formed during non-error reductions, memory bugs cannot "creep in" and manifest later, which would hinder the debugging process.

Lemma A.0.5 (Trace prefixes)**.** *If expression* **a** *is reduced before expression* **b***, then:*

1. *σ_a is a prefix of σ_b, denoted as $\sigma_a \sqsubseteq \sigma_b$.*

2. *$F_a \subseteq F_b$.*

Proof. By induction on the length of the reduction. □

Lemma A.0.6 (Order of events)**.** *Suppose the current state is $\Sigma_e = \langle H_e; F_e; \overline{S}_e; P_e; k_e; \sigma_e; f_e; e\rangle$, i.e., the redex is e, and there exists an event $\nu_a \in \sigma_e$. Then the expression a generating event ν_a has already been reduced (which implies a is reduced before e).*

Proof. By contradiction. □

Intuitively, Lemmas A.0.5 and A.0.6 show that reduction order is consistent with the order of corresponding events in the trace σ.

Lemma A.0.7. *Suppose we are in state*

$$\Sigma_e = \langle H_e; F_e; \overline{S}_e; P_e; k_e; \sigma_e; f_e; \mathsf{free}\ r_e\rangle$$

i.e., the redex is $e = \mathsf{free}\ r_e$, and we have $Bid(r_e) \notin Dom(H_e) \wedge (Bid(r_e), h) \notin Dom(F_e)$. Then there does not exist an event $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma_e$ such that $bid_a = Bid(r_e)$.

Proof. Suppose there exists an event $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma_e$ such that $bid_a = Bid(r_e)$. It is easy to see that the expression generating ν_a is $a = \mathsf{malloc}\ n_a$ and the reduction rule is [MALLOC]. $bid_a = Bid(r_e) \in Dom(H'_a)$ by rule [MALLOC]. Expression a is reduced before e by Lemma A.0.6. There are two subcases:

(a) there exists an expression $c = free\ r_c$ reduced after a and before e, where $Bid(r_c) = bid_a = Bid(r_e)$. Because expression c does not get stuck, the rule applied must be rule [FREE]. Then $(\mathit{Bid}(r_e), h) = (\mathit{Bid}(r_c), h) \in \mathit{Dom}(F'_c)$ by rule [FREE]. By Lemma A.0.5, $F'_c \subseteq F_e$, then $(\mathit{Bid}(r_e), h) = (\mathit{Bid}(r_c), h) \in \mathit{Dom}(F_e)$: a contradiction.

(b) there does not exist an expression $c = \mathsf{free}\ r_c$ reduced after a and before e, where $\mathit{Bid}(r_c) = \mathit{bid}_a = \mathit{Bid}(r_e)$, then $\mathit{Bid}(r_e) = \mathit{bid}_a \in \mathit{Dom}(H_e)$: a contradiction. □

Theorem A.0.8 (Soundness)**.** *Let the current state be* Σ*, where* $\Sigma \neq \mathit{Error}$*, the current trace be* σ *and the redex be* $\mathbf{e}$*. Suppose* $\mathbf{p}$ *is a bug detector, i.e., a predicate on* σ *and* $\mathbf{e}$*, and* [BUG-P] *is an error rule associated with the detector. If the machine's next state is an Error state (*[BUG-P] $\Sigma \longrightarrow \mathit{Error}$*) then the detector fires, i.e., predicate* $\mathbf{p}$ *is true.*

Proof. By induction on the reduction $\Sigma \longrightarrow \Sigma'$. Proceed by case analysis. Suppose the last reduction rule applied is R, there are nine cases for R.

case [BUG-UNMATCHED-FREE] **:**

The redex must be an expression b such that $b = \mathsf{free}\ r_b$, and $(\mathit{Bid}(r_b) \notin \mathit{Dom}(H_b) \wedge (\mathit{Bid}(r_b), h) \notin \mathit{Dom}(F_b)) \vee r_b \neq \mathit{Begin}(r_b)$ is true by the premise of the rule [BUG-UNMATCHED-FREE]. There are two subcases:

case $\mathit{Bid}(r_b) \notin \mathit{Dom}(H_b) \wedge (\mathit{Bid}(r_b), h) \notin \mathit{Dom}(F_b)$ holds **:**

By Lemma A.0.7, there does not exist an event $\nu_a = (k_a, \mathsf{malloc}, n_a, \mathit{bid}_a) \in \sigma_b$ such that $\mathit{bid}_a = \mathit{Bid}(r_b)$, thus $\neg\mathit{Allocated}(r_b)$ holds, hence $\mathbf{p}$ holds and detector [UNMATCHED-FREE] fires.

case $r_b \neq \mathit{Begin}(r_b)$ holds **:**

It is easy to see that the predicate for the unmatched free bug specified in Figure 3.2 fires if this error state is encountered; expression b corresponds to the detection point $\mathit{free}\ r_b$, and $\neg\mathit{Allocated}(r_b) \vee r_b \neq \mathit{Begin}(r_b)$ holds, hence $\mathbf{p}$ holds and detector [UNMATCHED-FREE] fires.

case [BUG-DOUBLE-FREE] **:**

The redex must be $b = \mathsf{free}\ r_b$, and $(\mathit{Bid}(r_b), \mathrm{h}) \in \mathit{Dom}(F_b)$ holds by the premise of the rule [BUG-DOUBLE-FREE]. By Lemma A.0.2, $F_0 = \emptyset$, hence there must exist an expression a, reduced before b, that changes F. The only rule which changes F is rule [FREE]. That is, there exists an expression $a = free\ r_a$, and $\nu_a = (k_a, \mathsf{free}, r_a, bid_a) \in \sigma'_a$, such that $(bid_a, h) = (\mathit{Bid}(r_b), h) \in \mathit{Dom}(F_b)$, and $bid_a \in \mathit{Dom}(H_a)$ by rule [FREE]. By Lemma A.0.5, $\sigma'_a \sqsubseteq \sigma_b$, so $\nu_a \in \sigma_b$.

Similarly, by Lemma A.0.2, $H_0 = \emptyset$; hence there must exist an expression c reduced before a and its corresponding reduction rule expands H (the only rule is [MALLOC]). That is, $c = \mathsf{malloc}\ n_c$, and $\nu_c = (k_c, \mathsf{malloc}, n_c, bid_c) \in \sigma'_c$, such that $bid_c = bid_a \in \mathit{Dom}(H_a)$. By Lemma A.0.5, $\sigma'_c \sqsubseteq \sigma_b$, so $\nu_c \in \sigma_b$.

It is easy to see that the detector [DOUBLE-FREE] specified in Figure 3.2 fires if this error state is encountered (expression b corresponds to the detection point $free\ r_b$, and the $\mathit{Allocated}(r_b)$ and $\mathit{Freed}(r_b)$ auxiliary predicates are true, because of the existence of event ν_c and ν_a in σ_b respectively, hence $\mathbf{p}$ holds).

case [BUG-DANG-PTR-DEREF] **:**

The redex must be $b = *r_b$, and $(\mathit{Bid}(r_b), h) \in \mathit{Dom}(F_b)$ holds by the premise of the rule [BUG-DANG-PTR-DEREF]. Similarly to the previous case, there exists an expression $a = \mathsf{free}\ r_a$ reduced before b, where $\nu_a = (k_a, \mathsf{free}, r_a, bid_a) \in \sigma_b$, and $bid_a = \mathit{Bid}(r_b)$, and there exists an expression $c = \mathsf{malloc}\ n_c$ reduced before a, where $\nu_c = (k_c, \mathsf{malloc}, n_c, bid_c) \in \sigma_b$, and $bid_c = \mathit{Bid}(r_b)$.

It is easy to see that the detector [DANGLING-POINTER-DEREF] specified in Figure 3.2 fires if this error state is encountered (expression b corresponds to the detection point $deref\ r_b$, and the $\mathit{Allocated}(r_b)$ and $\mathit{Freed}(r_b)$ auxiliary predicate is true because of the existence of event ν_c and ν_a in σ_b respectively, hence **p** holds).

case [BUG-DANG-PTR-DEREF2] **:**

The redex must be $b = r_b := v_b$, and $(Bid(r_b), h) \in Dom(F_b)$ by the premise of the rule [BUG-DANG-PTR-DEREF2]. The proof is similar to case [BUG-DANG-PTR-DEREF], hence the detector [DANGLING-POINTER-DEREF] fires.

case [BUG-NULL-PTR-DEREF] **:**

The redex must be $b = *r_b$, and $r = NULL$ by the premise of the rule [BUG-NULL-PTR-DEREF]. It is easy to see that **p** holds and the detector [NULL-POINTER-DEREF] specified in Figure 3.2 fires if this error state is encountered (expression b corresponds to the detection point $deref\ r_b$).

case [BUG-NULL-PTR-DEREF2] **:**

The redex must be $b = r_b := v_b$, and $r = NULL$ by the premise of the rule [BUG-NULL-PTR-DEREF2]. The proof is similar to case [BUG-NULL-PTR-DEREF] hence the detector [NULL-POINTER-DEREF] fires (expression b corresponds to the detection point $deref\ r_b$).

case [BUG-OVERFLOW] **:**

The redex must be $b = *r_b$, and $Bid(r_b) \in Dom(H_b) \wedge (r_b < Begin(r_b) \vee r_b \geq End(r_b))$ by the premise of the rule [BUG-OVERFLOW]. Similarly, we can prove that there exists an expression $a = \mathsf{malloc}\ n_a$ reduced before b, where $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma_b$, and $bid_a = Bid(r_b)$. Suppose there is an expression $c = \mathsf{free}\ r_c$ reduced after a and before b, where $\nu_c = (k_c, \mathsf{free}, r_c, bid_c) \in \sigma_b$, and $bid_c = bid_a$. Then $(Bid(r_b), h) = (bid_c, h) \in Dom(F'_c)$ by rule [FREE]. By

Lemma A.0.5, $(Bid(r_b), h) = (bid_c, h) \in Dom(F_b)$. Because state Σ_b is well-formed by Lemma A.0.3 and A.0.4, then $F_b \cap H_b = \emptyset$ by Definition A.0.1. Thus $Bid(r_b) = bid_c \notin Dom(H_b)$: a contradiction.

Thus, there is no such expression c and no corresponding event ν_c in σ_b.

It is easy to see that the detector [HEAP-BUFFER-OVERFLOW] specified in Figure 3.2 fires if this error state is encountered (expression b corresponds to the detection point $deref\ r_b$, and the $Allocated(r_b)$ holds because of the existence of event ν_a, and $\neg Freed(r_b)$ is true due to the non-existence of ν_c in σ_b. Meanwhile, $(r_b < Begin(r_b) \vee r_b \geq End(r_b))$ is trivially satisfied.

case [BUG-OVERFLOW2] **:**

The redex must be $b = r_b := v_b$, and $Bid(r_b) \in Dom(H_b) \wedge (r_b < Begin(r_b) \vee r_b \geq End(r_b))$ by the premise of the rule [BUG-OVERFLOW2]. The proof is similar to case [BUG-OVERFLOW] hence the detector [HEAP-BUFFER-OVERFLOW] fires (expression b corresponds to the detection point $deref\ r_b$).

case [BUG-UNINITIALIZED] **:**

The redex must be $b = *r_b$, and $Value(r_b) = junk$ by the premise of the rule [BUG-UNINITIALIZED]. By the definition of $Value(r)$, we get $Bid(r_b) \in Dom(H) \vee Bid(r_b) \in Dom(\overline{S})$. There are two subcases:

case $Bid(r_b) \in Dom(H)$ **:**

By Lemma A.0.2, $H_0 = \emptyset$, there must be some expression a reduced before b and its corresponding reduction rule expands H (the only rule is [MALLOC]). That is, there is $a = \mathsf{malloc}\ n_a$, and $\nu_a = (k_a, \mathsf{malloc}, n_a, bid_a) \in \sigma'_a$, such that $bid_a = Bid(r_b)$, and $H'_a[bid_a \mapsto (\boxed{\text{junk}}, n_a, k_a)]$. That is, $Value(r_b) = junk$

in state Σ'_a.

Suppose there exist intervening reductions which change the $Value(r_b)$ after a and before b. Because the only rule which can change the value of a memory block is rule [ASSIGN], all such expressions should have the form: $r := v$ and follow rule [ASSIGN]. Without loss of generality, let us consider one of such possible expression, e.g., $c = r_c := v_c$ and $\nu_c = (k_c, \mathsf{write}, r_c, v_c, f_c) \in \sigma'_c$, such that $r_c = r_b$. By rule [ASSIGN], $v_c \neq junk$, so $Value(r_b) \neq junk$ in state Σ_b: a contradiction.

Thus, there is no such expression c and no corresponding event ν_c in σ_b.

It is easy to see that the detector [UNINITIALIZED-READ] specified in Figure 3.2 fires if the uninitialized read error state is encountered (expression b corresponds to the detection point $deref_r\ r_b$, and $FindLast(\_, \mathsf{write}, r_b, \_,)$ is false because of the non-existence of event ν_c in σ_b; hence because $\neg FindLast(\_, \mathsf{write}, r_b, \_,)$ holds, $\mathbf{p}$ holds).

case $Bid(r_b) \in Dom(\overline{S})$ **:**

Similarly, there must be an expression $a = \mathsf{let}\ x = \mathsf{salloc}\ n_a\ \mathsf{in}\ e$, such that $bid_a = Bid(r_b)$, and $\overline{S}'_a[bid_a \mapsto (\boxed{\text{junk}}, n_a, k_a)]$; the proof is similar to the prior subcase.

□