A State Alteration and Inspection-based Interactive Debugger

Yan Wang
CSE Department, UC Riverside
wangy@cs.ucr.edu

Min Feng
NEC Laboratories America
mfeng@nec-labs.com

Rajiv Gupta Iulian Neamtiu
CSE Department, UC Riverside
{gupta,neamtiu}@cs.ucr.edu

***Abstract*—Despite significant advances in automated debugging, programmers still rely on traditional interactive debuggers (e.g., GDB) to find and fix bugs. While popular, these debuggers have major deficiencies: they do not guide the programmer in narrowing the source of error, and they only support a limited and low-level set of state-altering commands, hence semantic state alteration requires recompilation and reexecution. To address these shortcomings, we present an interactive debugger that combines capabilities that reduce debugging effort and increase debugging speed. The capabilities that yield these benefits include: *state alteration* commands for dynamically switching the directions of conditional branches and suppressing the execution of statements; *state inspection* commands including navigating and pruning dynamic slices; and *state rollback and checkpointing* commands to allow reexecution of the program from an earlier checkpoint. Our prototype is built on top of GDB using the Pin infrastructure; we also provide a GUI based on KDbg. Our experience shows that our debugger handles many kinds of real bugs effectively and efficiently.**

***Index Terms*—debugging, dynamic slicing, state alteration**

I. Introduction

Debugging is a long and laborious process which takes up to 70% of the total time of software development and maintenance [15]. To assist in the debugging process, programmers make use of an interactive debugger (e.g., GDB) whose use 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). 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* and then recompile and rerun the program to see how the program behavior is affected (e.g., commenting out suspicious statements [12]). To speed up this process, Visual Studio offers an Edit-and-Continue feature [26] 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.

978-1-4673-5739-5/13/$31.00 © 2013 IEEE

In this paper we present an interactive debugger which supports commands that 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 our debugger 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 our debugger 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.

In summary, our debugger supports new features that greatly improve bugfinding/bugfixing efficiency and speed up the iterative debugging process. Since it is built on top of GDB, all commands that are supported by GDB are still available. Some GDB commands have been enhanced and/or reimplemented for improved efficiency. Setting of breakpoints can be guided by program slices allowing the programmer to single-step from one statement in the slice to the next. Conditional breakpoints allow calls of library functions to be selectively captured. The overhead of the state rollback mechanism is reduced via incremental checkpointing. We have integrated the debugging back-end into the KDbg GUI, hence the new features are augmented by the GUI's visualization and navigation support. We have evaluated our debugger using five kinds of hard-to-find

TABLE I
MAJOR DEBUGGING COMMANDS

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 breakpoint`	conditionally capture memory bug 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

real bugs: double free, stack smashing, heap buffer overflow, dangling pointer dereference, and null pointer dereference. Our experience shows that our state alteration and inspection capabilities are effective and efficient.

II. Debugging Commands

We provide three kinds of debugging commands—**state alteration**, **state inspection**, and **state rollback**—listed in Table I. 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.

Our debugger 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., if the crash happens because a library call destroys an auto-maintained stack or heap. Our debugger 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. Our debugger 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. Our debugger 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 file. Navigating across source files is burdensome. Our debugger enables the programmer to visually navigate captured dependences and reason about the execution.

A. 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.

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) 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 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 [12]. 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 and recompile the program. For example, assume that the programmer has forgotten to use a guarding predicate around some statements, causing the program to crash. The programmer can try to suppress the unguarded statements based on his knowledge of the program. If desired results is observed, the programmer can then 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.

B. State Inspection Interfaces

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 idth 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 idth slice. The command `sbreak all id` inserts a breakpoint at each statement in the idth slice. Breakpoints for `sbreak` are triggered when specific execution instances are encountered.
- `sdelete id` is used to delete the idth 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 [7]. 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) Conditional Breakpoints: Existing debuggers (e.g., GDB) provide conditional breakpoints; however, the condition must be defined at source code level that is not available to programmers for library functions. Thus, conditional breakpoints must be set at each call site, that 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 argN|ret==value` triggers a breakpoint when the Nth 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 arg1==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 arg1==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.

C. 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 [34] 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

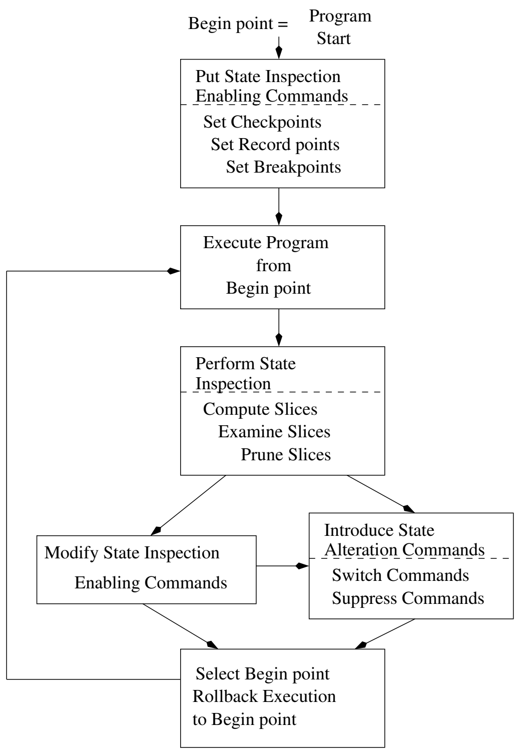

Begin point = Program Start
Put State Inspection Enabling Commands
Set Checkpoints
Set Record points
Set Breakpoints
Execute Program from Begin point
Perform State Inspection
Compute Slices
Examine Slices
Prune Slices
Modify State Inspection Enabling Commands
Introduce State Alteration Commands
Switch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

Fig. 1. Typical use of our debugger.

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.

III. Usage of Debugging Commands

In this section we first describe how different types of commands are used during the debugging process and then demonstrate their use in context of a set of hard-to-locate bugs from real programs.

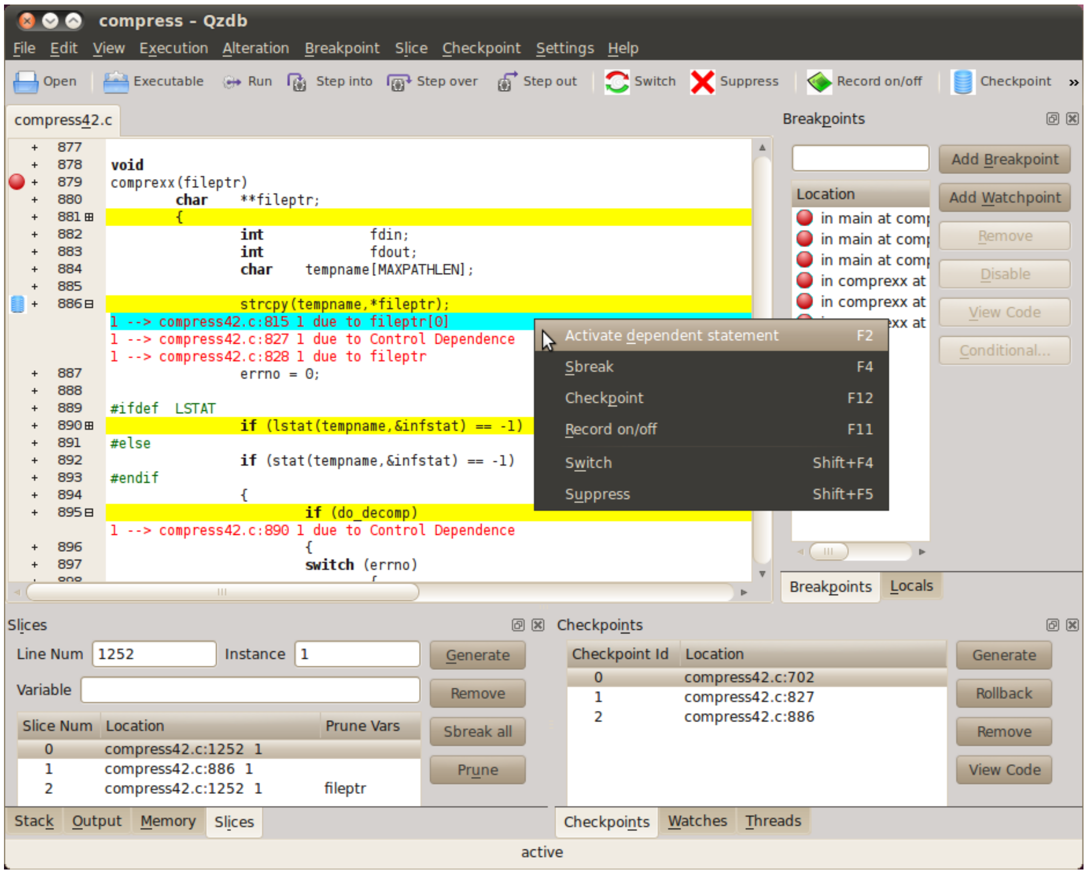
Figure 1 overviews the debugging process based upon the supported commands. Let us assume that the execution of the program has failed on an input. First, the programmer enters commands that will later allow detailed state inspection and program reexecution. Then the program is executed from the beginning 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.

Next we illustrate the process of finding a bug using our debugger. Figure 2 shows a stack buffer overflow bug (also referred to as stack smashing) in version 4.2.4 of the *ncompress* program. 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.

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 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 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 for the programmer to figure out which variables or memory regions are used at the crash point. If we use the statement



```
+ 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)
```

Slices

Line Num: 1252 Instance: 1

Variable:

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

Generate · Remove · Sbreak all · Prune

Checkpoints

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

Generate · Rollback · Remove · View Code

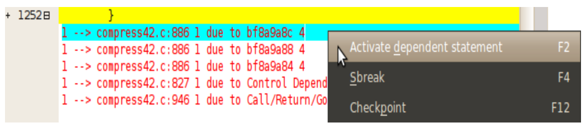
Fig. 2. The main window of our debugger.

line number (1252) and instance (1) as the slice criterion, the generated slice is as shown in Figures 2 and 3. 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, 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). Source code navigation along dependence edges can greatly enhance programmers' debugging efficiency.

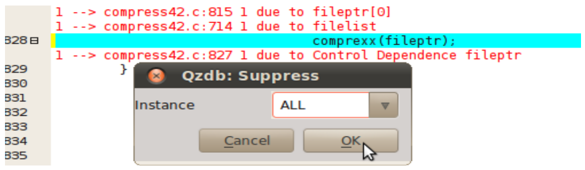
Following the dependence edges from the crash point of line 1252, the programmer knows that it is data-dependent on *strcpy* called at line 886 due to an unexpected write access to addresses `0xbf8a9a8c`, `0xbf8a9a88`, and `bf8a9a84` (see the first three dependence edges below line 1252 in Figure 3). 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 5), 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 4) or predicate switching (lines 825 or 827, Figure 5). 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



```
            }
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
```

Fig. 3. Slicing from the crash point.



```
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
        }
```

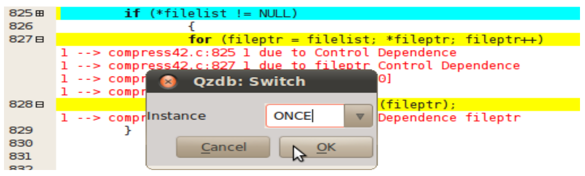
Fig. 4. Execution suppression.

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 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 slice 0 by 60% and dependence edges by 62%. Therefore, effective use of partial logging can greatly reduce slice sizes. The above process can be repeated for more complicated applications.

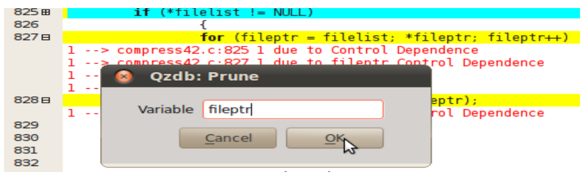
Additional Case Studies: Our case studies are based upon five different kinds of memory-related bugs listed in Table II, taken from BugNet [19]. The stack smashing bug was already discussed; we summarize the four other kinds of bugs due to space limitation.

Tidy-34132: This version 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 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 into 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. 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 into 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



```
            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
        }
```

Fig. 5. Predicate switching.



```
            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
```

Fig. 6. Pruning a slice.

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 arg1==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 [22]) 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 substitution.

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

TABLE II
OVERVIEW OF BENCHMARKS.

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

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.

Of course, 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 and helps fix the bug.

Bc-1.06: This version fails with a memory corruption error because a variable *v_count* is misused (instead of the correct *a_count*). The program performs dynamic array expansion, but due to the misuse, the heap object *arrays* is overflowed and the metadata maintained by the heap allocator is corrupted. Because of the metadata corruption, the programmer has difficulty getting any clue using standard debugging methods; with our debugger, checkpointing and reexecution combined with dynamic slicing reveal a hidden dependence in the array expansion and then via suppression the root cause is revealed.

Ghostscript-8.12: This version contains a dangling pointer dereference bug. Using checkpointing and suppression, users of our debugger quickly find out that suppressing function *i_free_object* causes the bug to go away; slicing the metadata that is supposed to be freed by *i_free_object* reveals an illegal write to it earlier on—the root cause of the bug.

Tar-1.13.25: This version dereferences a NULL pointer (variable *entry*) which causes a crash. Dynamic slicing of *entry* indicates earlier suspicious predicates; switching one of them suppresses the bug and reveals that the predicate is incorrect—the root cause of the bug.

```
istack.c:
025: AttVal *DupAttrs( TidyDocImpl* doc, AttVal *attrs) {
033:   *newattrs = *attrs;
034:   newattrs → next = DupAttrs( doc, attrs → next );  ...
       /* miss 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); }
```

Fig. 7. Double Free example.

In the above case studies we have shown the benefits of our approach in context of several widely-used applications containing different kinds of bugs.

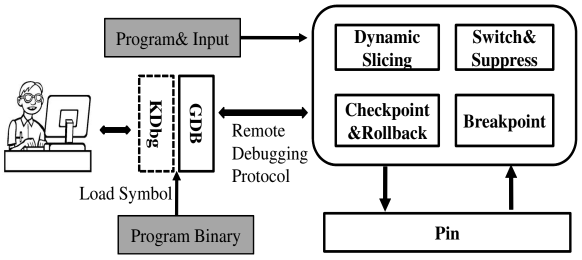


Fig. 8. Components of our debugger.

IV. IMPLEMENTATION

The prototype implementation of our interactive debugging strategy, shown in Figure 8, consists of GDB-based [7] and Pin-based [16], [33] components. The user interfaces with the GDB component via a command line interface or a KDbg [35] based graphical interface. The Pin-based component implements our new debugging commands. The GDB component communicates with the Pin-based component via the remote debugging protocol and it also interprets debug-information in the binary. 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.

TABLE III
RUN CHARACTERISTICS

Program	Test Case Description	Exec. Instr.	Baseline(sec)
ncompress	compress a folder(148KB)	10278947	0.33
tidy	check a HTML file(104 lines)	2125726	0.56
bc	interpret a source file(121 lines)	1846427	0.44
ghostscript	PS to PDF conversion(18KB)	3909749	0.47
tar	create an archive(789K)	4654490	0.51

TABLE IV
SLICING TIME AND SPACE OVERHEAD.

Program	Baseline Time (sec)	Pay-Once Time (sec) and Space (MB) Overhead					Slice Time Overhead (sec)		
		DU		CD		LP			
		Time	Space	Time	Space	Time	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

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. [31]. For accurately capturing dynamic control dependences, we use the online algorithm by Xin and Zhang [27]. The immediate postdominator information is extracted using Diablo [5].

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 [34] 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-call side-effects are detected by analyzing 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.

V. 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 II. Since our objective is to measure time and space costs, we used a passing test case to run each application to completion. Table III shows the run characteristics including number of executed instructions and the baseline running time under Pin without instrumentation. 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.

Slicing overhead. The time and space overhead for slicing are presented in Table IV. 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 preprocessing (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 IV 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 slice is not greatly dependent on the position of the slice criterion. Instead, it is dominated by the nature of slice criterion and program behavior. Most slice computations can be done in 1 min.,

TABLE V
TIME AND SPACE OVERHEAD: DU & CD

Program	MS/K instructions	KB/K instructions
ncompress	0.85	15.62
tidy	6.62	14.65
bc	4.45	13.88
ghostscript	3.38	13.10
tar	2.27	10.98
average	*3.51*	*13.65*

TABLE VI
CHECKPOINTING AND ROLLBACK TIME AND SPACE OVERHEAD.

Program	Num. of Checkpoints	Time (sec)	MS/K instructions	Space (KB)	KB/K instructions	Rollback Time (millisecond)
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

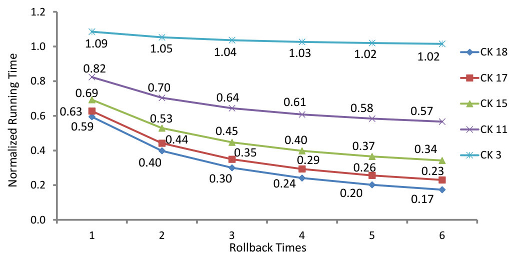


Fig. 9. Runtime savings due to rollback.

which is acceptable considering the large amount of time spent on debugging by programmers.

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 V, 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.

Checkpointing overhead. The time and space overhead of checkpointing are given in Table VI. 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 VI, 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 V, the time and space overhead per 1K instructions for incremental checkpointing is much lower. This is because only memory write instructions need to be instrumented for incremental checkpointing, while both memory and register read and write instructions need to be instrumented for recording.

Efficiency of state rollback. As mentioned in Section II-C, 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 pay 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 without rollback, are shown in Figure 9. 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.

TABLE VII
SUPPRESSION TIME OVERHEAD.

Program	Baseline (sec)	Suppress with Recording and Checkpointing (sec)	Suppress with Checkpointing Only (millisecond)
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%)

State alteration overhead. The time overhead of execution suppression is given in Table VII. 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 VII, 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 similarity in the implementation.

VI. RELATED WORK

Debugging Assistance. Dynamic slicing has been widely recognized as helpful for debugging [15], [31], [27], [29]. The Whyline [15] tool allows programmers to ask questions about program behavior and it responds with causal chains of events.

Techniques have been developed to automatically generate breakpoints based upon automated fault location [10], [29]. Chern et al. [3] improve breakpointing by allowing control-flow breakpoints. The *vsdb* interactive debugger uses symbolic execution to display all possible symbolic execution paths from a program point [9]. Coca [6] allows programmers to query the execution trace. However, none of these approaches take advantage of state alteration.

Fault Localization. State alteration has been used to automatically localize faults [30], [13], [2]. Zhang et al. [30] proposed predicate switching to constrain the search space of program state changes explored during bug location. Corrupted memory location suppression [13] attempts to identify the root cause of memory bugs by iteratively suppressing the cause of the memory failure. Chandra et al. [2] report repair candidates based on value replacement. Gu et al. [8] propose a bug query language allowing programmers to fix their bugs by referring to similar resolved bugs.

Delta debugging [4], [28] has been applied to automatically identify cause-effect chains [28] and recognize cause transitions [4]. Renieris and Reiss [25] identify faulty code by considering differences in statements executed by passing and failing runs. Tarantula [14] prioritizes statements based on their appearance frequency in failing runs versus passing runs. In general, these approaches rely on a large test suite that may not be available in practice. Limitations of these techniques (e.g., commands) motivate our work.

Memory-related Fault Detection, Tolerance, and Correction. Purify [11] and Valgrind [22] detect the presence of memory bugs via dynamic binary instrumentation. CCured [21] uses type inference to classify pointers and applies dynamic checks according to the classfication for memory safety. Rx [24] recovers from a crash by rolling back execution and reexecuting after changing the execution environment. AccMon [32] detects memory-related bugs by capturing violations of program counter based invariants. DieHard tolerates memory errors through randomized memory allocation and replication [1]. Exterminator dynamically generates runtime patches based upon runtime information [23]. Nagarakatte et al. use compile-time transformation for spatial [17] and temporal safety [18] of C.

VII. Conclusion

We have presented a novel interactive debugger that 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. Case studies on real reported bugs as well as performance evaluation demonstrate the effectiveness and efficiency of our debugger.

Acknowledgment

This research is supported by the National Science Foundation grant CCF-0963996 and a grant from Intel Corporation to the University of California, Riverside.

References

[1] Berger, E. D. and Zorn, B. G. DieHard: Probabilistic memory safety for unsafe languages. *PLDI*, pages 158–168, 2006.

[2] Chandra, S., Torlak, E., Barman, S., and Bodik, R. Angelic debugging. *ICSE*, pages 121–130, 2011.

[3] Chern, R. and De Volder, K. Debugging with control-flow breakpoints. *AOSD*, pages 96–106, 2007.

[4] Cleve, H. and Zeller, A. Locating causes of program failures. *ICSE*, pages 342–351, 2005.

[5] De Bus, B., De Sutter, B., Van Put, L., Chanet, D., and De Bosschere, K. Link-time optimization of arm binaries. *LCTES*, pages 211–220, 2004.

[6] Ducassé, M. Coca: an automated debugger for c. *ICSE*, pages 504–513, 1999.

[7] GDB 2011. http://www.gnu.org/software/gdb/.

[8] Gu, Z., Barr, E., and Su, Z. Bql: capturing and reusing debugging knowledge. *ICSE*, pages 1001–1003, 2011.

[9] Hähnle, R., Baum, M., Bubel, R., and Rothe, M. A visual interactive debugger based on symbolic execution. *ASE*, pages 143–146, 2010.

[10] Hao, D., Zhang, L., Zhang, L., Sun, J., and Mei, H. Vida: Visual interactive debugging. *ICSE*, pages 583–586, 2009.

[11] Hastings, R. and Joyce, B. Purify: Fast detection of memory leaks and access errors. *USENIX Winter Tech. Conf.*, pages 125–136, 1992.

[12] IBM tutorial on debugging 2011. IBM tutorial on debugging–Debugging using comments.

[13] Jeffrey, D., Gupta, N., and Gupta, R. Identifying the root causes of memory bugs using corrupted memory location suppression. *ICSM*, pages 356–365, 2008.

[14] Jones, J. A. and Harrold, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. *ASE*, pages 273–282, 2005.

[15] Ko, A. and Myers, B. Debugging reinvented:asking and answering why and why not questions about program behavior. *ICSE*, pages 301 –310, 2008.

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

[17] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Softbound: highly compatible and complete spatial memory safety for c. *PLDI*, 2009.

[18] Nagarakatte, S., Zhao, J., Martin, M. M., and Zdancewic, S. Cets: compiler enforced temporal safety for c. *ISMM*, pages 31–40, 2010.

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

[20] ncompress 2011. Ncompress: a fast, simple lzw file compressor. http://ncompress.sourceforge.net/.

[21] Necula, G. C., McPeak, S., and Weimer, W. CCured: type-safe retrofitting of legacy code. *POPL*, pages 477–526, 2002.

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

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

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

[25] Renieris, M. and Reiss, S. Fault localization with nearest neighbor queries. *ASE*, pages 30–39, 2003.

[26] Visual Studio Debugger 2011.

[27] Xin, B. and Zhang, X. Efficient online detection of dynamic control dependence. *ISSTA*, pages 185–195, 2007.

[28] Zeller, A. Isolating cause-effect chains from computer programs. *FSE*, pages 1–10, 2002.

[29] Zhang, C., Yan, D., Zhao, J., Chen, Y., and Yang, S. BPGen: an automated breakpoint generator for debugging. *ICSE*, 2010.

[30] Zhang, X., Gupta, N., and Gupta, R. Locating faults through automated predicate switching. *ICSE*, pages 272–281, 2006.

[31] Zhang, X., Gupta, R., and Zhang, Y. Precise dynamic slicing algorithms. *ICSE*, pages 319–329, 2003.

[32] Zhou, P., Liu, W., Fei, L., Lu, S., Qin, F., Zhou, Y., Midkiff, S. P., and Torrellas, J. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. *MICRO*, pages 269–280, 2004.

[33] Lueck, G., Patil, H., Pereira, C. PinADX: an interface for customizable debugging with dynamic instrumentation. *CGO*, pages 114–123, 2012.

[34] King, S. and Dunlap, G. and Chen, P. Debugging operating systems with time-traveling virtual machines. *ATEC*, pages 1–1, 2005.

[35] Kdbg homepage http://www.kdbg.org/.