Dynamic Slicing Long Running Programs through Execution Fast Forwarding

Xiangyu Zhang   Sriraman Tallam   Rajiv Gupta
Department of Computer Science
The University of Arizona
Tucson, Arizona 85721
{xyzhang,tmsriram,gupta}@cs.arizona.edu

ABSTRACT

Fixing runtime bugs in long running programs using trace based analyses such as dynamic slicing was believed to be prohibitively expensive. In this paper, we present a novel *execution fast forwarding* technique that makes this feasible. While a naive solution is to divide the entire execution by checkpoints, and then apply dynamic slicing enabled by tracing to one checkpoint interval at a time, it is still too costly even with state-of-the-art tracing techniques. Our technique is derived from two key observations. The first one is that long running programs are usually driven by events, which has been taken advantage of by checkpointing/replaying techniques to deterministically replay an execution from the event log. The second observation is that all the events are not relevant to replaying a particular part of the execution, in which the programmer suspects an error happened. We develop a slicing-like technique that can be used to prune irrelevant events from the event log. Driven by the reduced log, the replayed execution is now traced for fault location. This replayed execution has the effect of fast forwarding, i.e the amount of executed instructions is significantly reduced without losing the accuracy of reproducing a failure. Our evaluation shows that skipping irrelevant events can reduce the space requirement for dynamic slicing by factors ranging from 72 to 44490. We also describe how checkpointing and tracing enabled dynamic slicing are combined, which we believe is the first attempt to integrate these two techniques. Finally, the dynamic slices of a set of reported bugs for long running programs are studied to show the effectiveness of dynamic slicing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Testing tools, Tracing*; D.3.4 [**Programming Languages**]: Processors—*Debuggers*

General Terms

Algorithms, Measurement, Reliability, Verification

Keywords

debugging, checkpointing, event logging, replay, data slicing

1. INTRODUCTION

During the procedure of debugging, it is often the case that the programmer is interested in a small part of the entire execution. How to get to this region quickly has been haunting researchers since debugging long running programs became an issue. The traditional debugging tactics, such as iteratively setting breakpoints and then restarting the program, hardly work because the reexecutions consume enormous amount of time. More sophisticated methods to tackle this problem include *tracing* and *checkpointing/replaying*.

Tracing is a technique with a long history. It was invented for the purpose of replaying an execution. More and more applications have been developed such as performance analysis, software reliability, software understanding, and compiler optimizations. While in a classical debugging procedure breakpoints are set and the program is reexecuted many times till the bug is located, in tracing the execution traces are usually collected once and then are analyzed multiple times starting from selected points. Furthermore a wide variety of heavy duty analyses can be performed on traces efficiently. As a result, software errors become much more recognizable if appropriate traces are gathered. For example, dynamic slicing, proposed by Korel and Laski [5], is a tracing based technique to help programmers in the process of debugging. The dynamic slice of a value computed at an execution point includes all those executed statements which were directly or indirectly involved in computation of the value. Our prior work [23, 3, 24, 25] has demonstrated that dynamic slicing is quite effective in automatically isolating the cause effect chain from the root cause to the failed point. Unfortunately, tracing based techniques do not scale to long executions even though state-of-the-art techniques can achieve the space efficiency of 0.1 - 4 bits per instruction [22, 1]. A simple task as starting Mozilla and browsing a html page may create traces with the size of a few Gigabytes.

Checkpointing/replaying is a very attractive technique, the merit of which is the capability of replaying from the intermediate points of an execution once checkpoints are created. It was invented to facilitate debugging parallel and distributed programs [11, 20]. It quickly gained popularity in debugging general applications [14, 15]. A lot of research has been carried out on how to reduce its cost [19, 8] and how to improve its usability [17]. Most of the existing checkpointing techniques focus on how to faithfully replay an execution. They rarely discuss what to do with replayed executions or simply suggest that replayed executions can be debugged with general debuggers such as gdb. However, these debuggers are usually much less powerful than tracing based tools.

SIGSOFT'06/FSE-14, November 5–11, 2006, Portland, Oregon, USA.
Copyright 2006 ACM 1-59593-468-5/06/0011 ...$5.00.

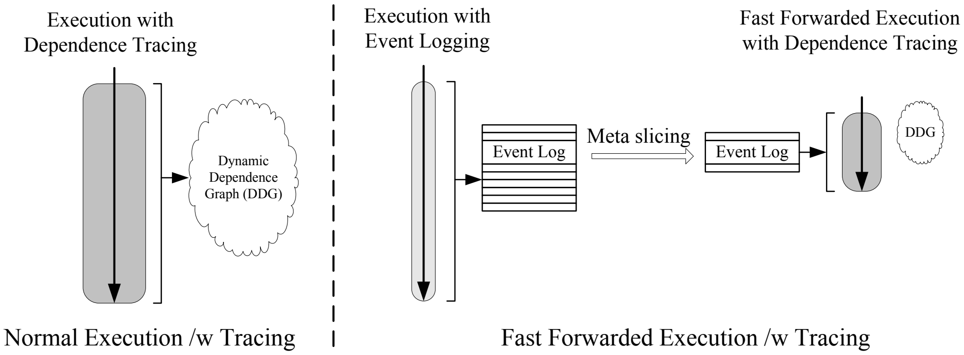


Figure 1: Execution Fast Forwarding.

Our goal is to apply dynamic slicing, a tracing based technique, to long running programs. A natural question to ask is ”*Can we combine tracing and checkpointing?*”. It seems tracing and checkpointing are complementary. Checkpoints divide the whole execution into intervals. Tracing can be applied to one interval at a time, usually the one that interests the programmer. However, this solution is not as simple as it appears for two reasons. First, tracing requires instrumenting the original program. There are two kinds of instrumentation techniques – static and dynamic. Static instrumentation, in which a program is instrumented by a compiler, introduces non-trivial execution overhead as tracing cannot be easily turned off. Dynamic instrumentation adaptively instruments a program. It can easily switch from executing the original code to executing the instrumented code or vice versa. A dynamic instrumentation engine usually resides in an application process's virtual space and manipulates the virtual memory intensively such that the status of the application process is substantially mixed with the instrumentation engine's status. While checkpoints are often produced by taking snapshots of the virtual memory, it becomes hard to discretely checkpoint the application process. Second, tracing can handle executions of up to a few seconds given the speed and storage capacity of modern workstations. Since checkpointing usually produces virtual memory snapshots with the size of a few Megabytes, it is not something that we can afford to perform every second. Checkpoints are usually created in an interval of, more or less, minutes. The gap between seconds and minutes suggests that it is still too costly to trace an entire checkpoint interval.

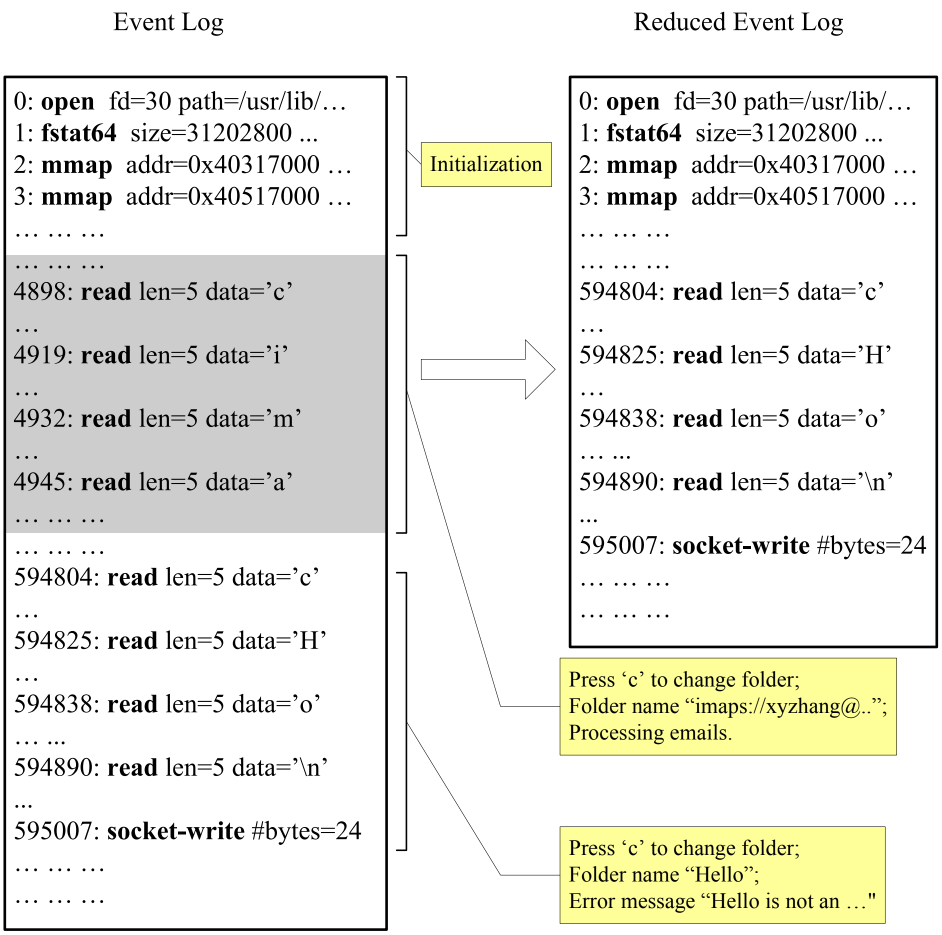
In this paper, we present a novel *execution fast forwarding (EFF)* technique that fills the gap between tracing and checkpointing. It enables dynamic slicing of long executions. Figure 1 illustrates the basic idea. The left part illustrates that an execution, or part of an execution delimited by checkpoints, is usually heavily instrumented for the purpose of dependence tracing. The heavy instrumentation introduces very high runtime overhead and constructs a huge dependence graph, which makes it impractical if the execution gets long. In the right part a fast forwarding technique takes advantage of the characteristics of many long running programs – being driven by events. More precisely, it first collects a full event log from the original execution. Next, given a specific part of the execution that the programmer wants to replay, a meta slicing technique, which is analogous to dynamic slicing but performed on logged events instead of executed instructions, is applied to prune the events irrelevant to the replay of desired execution region. The reduced event log is used to drive the replay, which is also called *the fast forwarded execution*. Compared to the original run, the fast forwarded execution is much smaller as the volume of events passed to the program is significantly lower. As a result, a smaller dependence graph is generated that can be collected through tracing. The contributions of our paper are summarized as follows.

- We propose a solution to debugging long running programs, which consists of the steps of checkpointing and logging a long execution, reducing the log file, replaying the execution with the reduced log, and dynamic slicing during replay.
- We develop a novel EFF technique that performs meta slicing on an event log to eliminate the events that are not relevant to replaying a specific part of execution. The reduced event log is used to drive a replayed execution to achieve the effect of fast forwarding.
- To implement the EFF technique, we show how to combine tracing and logging/checkpointing. Given the strengths of these techniques, we believe integrating them has very high potential to impact the existing debugging procedures.
- As the ultimate goal of EFF, dynamic slicing is applied on a set of long running programs, which was not possible previously due to its extremely high cost. The results strongly support our claim in the prior work – *dynamic slicing is very effective in isolating the cause effect chain from the root cause to the failure* [23, 25].

The remainder of the paper is organized as follows. In section 2 we describe the EFF technique in detail. The system, which is an integration of EFF, tracing and checkpointing, is introduced in section 3. The results of our experiments are presented in section 4. In section 5 we studied the effectiveness of dynamic slicing on long running programs. Related work is discussed in section 6 and conclusions are given in section 7.

2. EXECUTION FAST FORWARDING

Often when a program runs for a long time, it is not because the program performs a very long and complicated task. Instead, it is often because the program processes a long sequence of simple tasks. For example, programs processing streaming data such as audio, video, and data packets usually carry out the same computation (e.g., the FFT transformation) on a sequence of packets. The computation on each packet tends to be relatively lightweight and independent from the computation for other packets. Programs that require user interactions display similar properties: these programs spend most of their execution time in handling user actions and the computation dedicated for each user action is usually simple. Server programs deal with thousands of requests, most of which set



Event Log

```
0: open  fd=30 path=/usr/lib/…
1: fstat64  size=31202800 ...
2: mmap  addr=0x40317000 …
3: mmap  addr=0x40517000 …
… … …
… … …
4898: read len=5 data=’c’
…
4919: read len=5 data=’i’
…
4932: read len=5 data=’m’
…
4945: read len=5 data=’a’
… … …
… … …
594804: read len=5 data=’c’
…
594825: read len=5 data=’H’
…
594838: read len=5 data=’o’
… ...
594890: read len=5 data=’\n’
...
595007: socket-write #bytes=24
… … …
… … …
```

Reduced Event Log

```
0: open  fd=30 path=/usr/lib/…
1: fstat64  size=31202800 ...
2: mmap  addr=0x40317000 …
3: mmap  addr=0x40517000 …
… … …
… … …
594804: read len=5 data=’c’
…
594825: read len=5 data=’H’
…
594838: read len=5 data=’o’
… ...
594890: read len=5 data=’\n’
...
595007: socket-write #bytes=24
… … …
… … …
```

Initialization

Press ‘c’ to change folder;
Folder name “imaps://xyzhang@..”;
Processing emails.

Press ‘c’ to change folder;
Folder name “Hello”;
Error message “Hello is not an …"

Figure 2: Getting the same warning message by replaying the reduced log for Mutt 1.4.2.1i. The numbers mean the byte positions of the corresponding events in the log.

off simple computations such as reading a file or retrieving a piece of data from a database. A common feature of these programs is that *they are driven by events*. Events divide a whole execution into small tasks, each one of which corresponds to handling some event. An event is defined as one interaction between the application and the OS. The interaction could be in the forms of: system calls such as *open*, *read*, and *mmap2*; asynchronous or synchronous signals such as *kill* and *segfault* etc. These events are used to provide OS services, such as reading/writing a file/socket, to the application program or to notify something has happened.

The EFF technique is derived from the following observation – *all events do not need to be replayed in order to replay a particular part of execution*. Given an execution that is driven by events, we may be able to shrink the replayed execution, and yet reproduce the desired part, if we can prune the irrelevant events.

Figure 2 presents a motivating example. In the original run, the key ’*c*’ was first pressed in order to change the folder name after Mutt, a text based mail user agent, was started; string ”*imaps://xyzhang@email.cs.arizona.edu/inbox*” was typed in as the email account, which was followed by the password. After logging in the account, a couple of email messages were accessed, then ’*c*’ was typed again, and string ”*Hello*” was provided as the new folder name. Since ”*Hello*” was not a valid folder name, a warning message was printed on the screen. The events were logged in a file as shown on the left hand side of the figure. The first few thousands of events represent the startup phase of the execution, which mainly performs: loading of dynamic libraries, allocating virtual memory, and initializing the program state. The shaded events starting from byte position 4898 to position 594803 correspond to the execution related to accessing the email account. Events starting from 594804 contribute to entering the invalid folder name and the warning message was printed by the event at 595007. Let us assume the programmer is interested in reproducing the warning message. Apparently, replaying the entire execution with the full log is an option but not the optimal one. For the event at 595007 to be correctly replayed, we need to replay events at 594804, 594825, ..., 594890. Events from 4898 to 594803 are actually *irrelevant* to replaying the event at 595007. We construct a new log by removing all the irrelevant events and then drive the replay with the reduced log. The same warning message is successfully reproduced. The execution was actually *fast forwarded* to the desired point by skipping the irrelevant part.

The EFF technique poses two challenges. The first challenge is the identification and removal of irrelevant events. The second challenge is to develop a replay mechanism that works with a reduced event log. The following subsections describe how we handle these issues.

2.1 Event Dependence Graph

In dynamic slicing, given a value that is observed to be incorrect by the programmer (incorrect value may correspond to an incorrect output or a value that causes the program to crash), a set of executed statements that contributed to the incorrect value are computed as its dynamic slice. The executed statements not in the dynamic slice are not relevant to the investigated value. An analogous technique can be applied to executed events to identify the set of relevant events for replaying a given execution region.

Computation of dynamic slices normally consists of two steps: building the dynamic dependence graph (DDG) for a failed execution (where dependences include both data and control dependences); and then traversing the dynamic dependence graph to compute the dynamic slice of the wrong value. To simplify the description, we assume the execution starts from the beginning. We will discuss how to deal with executions starting from checkpoints in later sections.

DEFINITION 1. *The* **Dynamic Dependence Graph** *of a program run, DDG(N, E), consists of a set of nodes N and a set of directed edges E where: each node $n_i \in N$ corresponds to the i^{th} execution instance of statement n in the program; and each edge $m_j \rightarrow n_i \in E$ corresponds to a dynamic data dependence, dynamic control dependence, or potential dependence of the i^{th} execution instance of statement n on the j^{th} execution instance of statement m.*

In a DDG, an executed statement is abstracted as $S_j(U, D)$ which denotes the j^{th} instance of statement S and where U denotes the set of values used by S_j and D denotes the set of values defined. For example, the execution of statement "*store* r_1, $[r_2]$" can be abstracted as "...($U = \{r_1, r_2\}, D = \{[r_2]\}$", in which $[r_2]$ represents the memory location addressed by r_2. A data dependence exists between two executed statements if the U set of one statement overlaps the D set of the other. A control dependence is introduced if the execution of one statement depends on the values in D of the other statement, usually a predicate statement. One executed statement S_j *potentially depends* on another executed statement, usually a predicate, if and only if the value of the executed statement could have changed if the predicate had taken a different branch. More details about potential dependence can be found in [4, 23].

We already discussed how an executed statement is abstracted. As an event usually corresponds to multiple executed statements, it is important to understand how we deal with events during DDG construction. Since system calls are usually handled inside the OS kernel, a tracing engine which runs in the application space is not able to trace into the kernel. Hence the dependences within a system call are not captured. Our solution is to summarize the execution of a system call, or an event, into the same abstraction, $E_j(U, D)$, according to the specifications of events. For instance, event "*n=read(fd, Buf, size)*" can be abstracted as "...($U =$ $\{$ *fd, seek_pointer(fd), size, Buf* $\}$, $D = \{$ *seek_pointer(fd), Buf[0], Buf[1], ... Buf[n-1]* $\}$. Note that only the first n elements of *Buf* are defined according to the specification of event *read*. This event both defines and uses the seek pointer of file *fd*.

An analogous dependence graph, *Event Dependence Graph (EDG)*, can be constructed to reveal the dependences between events, which can be later on used to prune irrelevant events.

DEFINITION 2. *The* **Event Dependence Graph** *of a program run, EDG(N, E), consists of a set of nodes N and a set of directed edges E where: each node $n_i \in N$ corresponds to the i^{th} execution instance of event n in the program; and each edge $m_j \rightarrow n_i \in E$ denotes that there exists a dependence path from m_j to n_i, and there are no other executed events than m_j and n_i on the path.*

Figure 3 presents an example to illustrate DDG and EDG. The left hand side presents the DDG. Statement executions 2_1 and 4_1 data depend on 1_1 because they use the file descriptor defined at 1_1. 4_1 data depends on 2_1 because 2_1 changes the file seek pointer. The graph on the right hand side shows the EDG. Event execution $E3_1$ depends on $E2_1$ because of the dependence path $2_1 \rightarrow 4_1$. Event execution $E4_1$ depends on $E2_1$ due to the dependence path $2_1 \rightarrow 3_1 \rightarrow 5_1 \rightarrow 6_1$. Note that the *read* events $E2$ and $E3$ are considered as different events because they occur at different program locations.

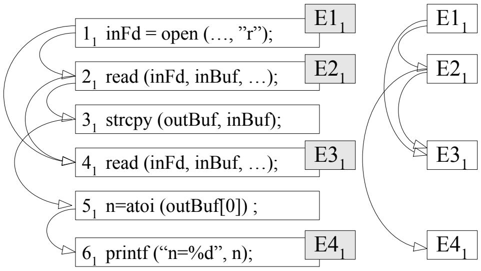

1_1 inFd = open (…, ”r”);
E1_1
2_1 read (inFd, inBuf, …);
E2_1
3_1 strcpy (outBuf, inBuf);
4_1 read (inFd, inBuf, …);
E3_1
5_1 n=atoi (outBuf[0]) ;
6_1 printf (“n=%d”, n);
E4_1
E1_1
E2_1
E3_1
E4_1

Figure 3: An example of Dynamic Dependence Graph (DDG) and Event Dependence Graph (EDG).

Control dependence between statements can also lead to dependence between events as demonstrated by another example in Figure 4, where event $E3_1$ depends on event $E2_1$ as the result of 30_1 control depending on 21_1 and 21_1 data depending on 20_1. The dependence between $E2_1$ and $E3_1$ belongs to control dependence as the execution of $E3_1$ is due to the result of $E2_1$. However, in EDGs we do not distinguish data dependence and control dependence edges.

Precisely constructing an EDG requires accurately tracing each data/control/potential dependence. According to our experience, exactly tracing each data/control dependence on the fly triggers a slow down of up to two orders of magnitude. Potential dependence is even more expensive to trace hence it is usually implemented as a post-mortem analysis. Thus, building a precise EDG is a luxury that becomes worthy only when the cost can be amortized by a large number of replays. Otherwise, programmers would prefer to replay the entire log, which is equivalent to doubling the execution time, rather than endure the two orders of magnitude slow down in the first place and attain speed up in replays later on. To address this issue, we have to be conservative by constructing an approximate EDG, in which one event depends on the other if and only if they are related by a *static* dependence path. In other words, we only demand a static dependence graph, instead of a dynamic one, together with the event log to build an approximate EDG. The only runtime overhead is caused by event logging, which is significantly lower than tracing each dependence. Because dependences between events are usually simpler than dependences between normal statements, which can be highly complicated due to pointer aliasing, being conservative in EDG construction introduces much less imprecision compared to being conservative in building DDG.

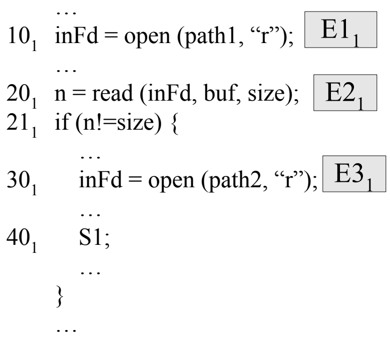
2.2 Meta Slicing on Event Log

Similar to dynamic slicing, given an EDG and an event, which the programmer wants to reproduce, meta slicing on the EDG computes the set of events that are needed in order to replay the given event.

DEFINITION 3. *Given EDG(N, E), an event dependence graph, the* **Meta Slice** *of $e_i \in N$ denoted by MS(e_i) is the subgraph of EDG(N, E) which includes e_i as well as all other nodes and edges* *from which e_i is reachable, i.e.*

$$MS(e_i) = (\{e_i\}, \{e|e = m_j \rightarrow e_i \in E\}) \cup \bigcup_{\forall m_j \rightarrow e_i} MS(m_j)$$

For example in Figure 3, MS$(E4_1) = \{E1_1, E2_1, E4_1\}$. Note that we ignore the edges in MS for simplicity. We need to replay $E1_1$, which opens a file, and $E2_1$, which reads some data from the file, in order to correctly replay $E4_1$, which prints some value resulted from computation over the input data. In Figure 4, MS$(E3_1) = \{E1_1, E2_1, E3_1\}$. $E2_1$ has to be replayed otherwise the control would not flow to $E3_1$.



```
       …
10_1   inFd = open (path1, "r");   E1_1
       …
20_1   n = read (inFd, buf, size);   E2_1
21_1   if (n!=size) {
         …
30_1     inFd = open (path2, "r");   E3_1
         …
40_1     S1;
         …
       }
       …
```

Figure 4: Another example of Event Dependence Graph.

We have discussed how to find the set of relevant events in order to replay a given event. However, in reality it could be a specific executed statement n_j that the programmer wants to replay. In this case, we need to find out the set of closest events reachable from n_j in the DDG, denoted as ECut(n_j), and then compute meta slices on these events. For example in Figure 4, ECut$(40_1) = \{20_1\}$, the corresponding meta slice MS$(20_1) = \{10_1, 20_1\}$. Intuitively, both $E1_1$ and $E2_1$ need to be replayed in order to replay statement $S1$.

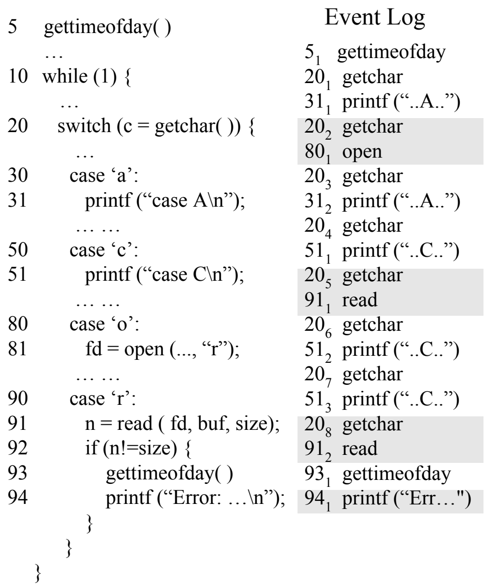
THEOREM 1. *The events in MS(ECut(n_j)) are sufficient to replay n_j.*

Proof. Let us assume there is an event e_x not in MS(ECut(n_j)), and e_x needs to be replayed in order to replay n_j. We infer there must exist an executed statement, event or non-event, m_i s.t. n_j is reachable from m_i and m_i is reachable from e_x. In other words, n_j directly/indirectly depends on m_i and m_i directly/indirectly depends on e_x. Otherwise, executing n_j would not require executing e_x. If there are no executed events along the path $e_x \rightarrow m_i \rightarrow n_j$ other than e_x, $e_x \in$ ECut(n_j), which is contradictory to the assumption; if there exists some executed event other than e_x along the path, let us assume e'_y is the executed event closest to n_j on the path s.t. $e'_y \in$ eCut(n_j), $e_x \in$ MS(e'_y) according to the definitions of EDG and meta slicing. It is a contradiction to the assumption. This completes the proof.

Note that in practice ECut has to be conservatively computed as we do not have dynamic dependence information. Our experience shows that this is not a problem because the events in ECut tend to be very close to the desired statement instance in the dependence graph such that very limited number of spurious dependences are brought in during the computation of ECut.

2.3 Replaying with A Reduced Event Log

We have described how meta slicing can be applied to identify a set of events that are relevant to replaying a given part of execution. However, meta slicing is not yet an ultimate solution. Even though the events in a meta slice are sufficient to replay a desired part of execution, the meta slice per se is often not a legitimate log to drive an execution. For example, in Figure 3, MS$(E4_1) = \{E1_1, E2_1, E4_1\}$. Replaying with the meta slice fails because $E3_1$ was expected when the control flows to statement 4_1. This suggests that some events, even though irrelevant to replaying the desired part of the execution, cannot be pruned due to the control flow structure. In this subsection, we are going to describe how an event log is reduced with regard to a meta slice and the intrinsic control flow structure of an application.



```
5    gettimeofday( )
     …
10   while (1) {
       …
20     switch (c = getchar( )) {
         …
30       case 'a':
31         printf ("case A\n");
         … …
50       case 'c':
51         printf ("case C\n");
         … …
80       case 'o':
81         fd = open (..., "r");
         … …
90       case 'r':
91         n = read ( fd, buf, size);
92         if (n!=size) {
93            gettimeofday( )
94            printf ("Error: …\n");
           }
       }
   }
```

```
Event Log
5_1   gettimeofday
20_1  getchar
31_1  printf ("..A..")
20_2  getchar
80_1  open
20_3  getchar
31_2  printf ("..A..")
20_4  getchar
51_1  printf ("..C..")
20_5  getchar
91_1  read
20_6  getchar
51_2  printf ("..C..")
20_7  getchar
51_3  printf ("..C..")
20_8  getchar
91_2  read
93_1  gettimeofday
94_1  printf ("Err…")
```

Figure 5: An example on reducing the event log. The shaded events are those in MS(94_1).

Before we present the algorithm, let us first study an example that clearly explains how it is made possible to reduce a log without losing validity. In Figure 5, the program displayed in the left column takes user commands from *stdin*. Different actions are taken based on different commands. For instance, messages are printed on the screen if 'a'/'c' is pressed; a file is opened if 'o' is pressed; the opened file is read if 'r' is read; if the data read does not match the size required, an error message is delivered. The event log for a particular execution is presented in the right column. During the execution, a file is opened and then read for twice; the second read does not satisfy the size wanted such that an error message is printed at 94_1; in between of these events, a number of events happen as the results of 'a'/'c' being pressed. Let us assume 94_1 is the event we want to replay. MS(94_1) is denoted as the shaded events in the log. Apparently, the meta slice is not legitimate for replay as event 5_1(*gettimeofday*), which is not in the meta slice, is expected at the beginning of the replayed execution. While 5_1 is not removable, events 20_1 and 31_1 can be removed without any problem. The important observation here is that 20_2 and 20_1 are *compatible* and thus 20_2 can be moved up to replace 20_1 such that the event in between, 31_1, is pruned.

DEFINITION 4. *An event execution e_i is compatible with another event execution e_j iff their calling contexts are identical and they occur at the same program point.*

Here the calling context of e_i represents the application's call stack when e_i is executed. All the events 20_x in Figure 5 are compatible to each other. This example suggests we are able to alter a replayed execution by replacing an event with its compatible peer.

The algorithm to reduce a log given a meta slice is presented as follows. Get_next_event() gets the next event from the log file; get_next_marked_event() gets the next event belongs to the meta slice, which we assume is precomputed, in the log file. These two methods share the same file seek pointer, which can be set by set_file_pointer(...).

```
Input:  the original log Log
Output:  the *reduced* log RLog
Initialize: RLog $\longleftarrow \phi$
while ($e_m$=get_next_marked_event(Log))!=EOF do
    $e$=get_next_event(Log)
    for each $e_t$ from $e$ to $e_m$ in Log do
        if $e_t$.context $\equiv$ $e_m$.context then
            goto $L_1$
        endif
        Rlog $\longleftarrow$ Rlog $\cdot$ $e_t$
    endfor
$L_1$:
    Rlog $\longleftarrow$ Rlog $\cdot$ $e_m$
    set_file_pointer(Log, $e_m$)
endwhile
```

The basic idea of the algorithm is that given a marked event e_m, an event in the meta slice, we find the earliest compatible event e_t in between e and e_m such that moving e_m up to replace e_t maximizes the savings. All the events between e and e_t including e are copied to the new log to satisfy the control flow structure confinement. The events between e_t and e_m are discarded.

Table 1 presents the reduction procedure of the example in Figure 5. As shown in the table, during iteration one, 5_1 is the first event retrieved from the log, and 20_2 is the first marked event. 20_2 can be moved up to replace 20_1 such that 5_1 and 20_2 are the two events appended to the new log. During the second iteration, 80_1 is the next event and also the next marked event such that it is simply copied to the new log. In iteration three, moving 20_5 up to replace 20_3 results in cutting the events from 20_3 to 50_1. The final reduced log is shown in the last row of the table. The reduce log can be used to drive the replayed execution to reproduce the error message at 94_1.

Table 1: Computation table for figure 5.

Iteration	e	e_m	RLog
1	5_1	20_2	5_1 20_2
2	80_1	80_1	5_1 20_2 80_1
3	20_3	20_5	5_1 20_2 80_1 20_5
4	91_1	91_1	5_1 20_2 80_1 20_5 91_1
5	20_6	20_8	5_1 20_2 80_1 20_5 91_1 20_8
6	91_2	91_2	5_1 20_2 80_1 20_5 91_1 20_8 91_2
7	93_1	94_1	5_1 20_2 80_1 20_5 91_1 20_8 91_2 93_1 94_1

2.4 Dynamic Slicing during Replay

Dynamic slicing was believed to be too expensive to apply for long executions. With sophisticated compression techniques [22] we can achieve the space efficiency of four bits per executed instruction, which is still not powerful enough for executions that run for minutes, hours, or days. The EFF technique can reproduce a failure without going through most of the irrelevant part of the execution. As a result, dynamic slicing becomes feasible for fast forwarded executions. According to our previous study [23], dynamic data slicing, in which slices are computed by considering only data dependence, is quite effective for memory type of bugs. Therefore, we only compute data slices in this paper due to the fact that crashes are usually the type of bugs reported for long running programs. In the remaining part of the paper, we mean dynamic data slices when we mention dynamic slices. Note that dynamic slicing in this phase is different from meta slicing mentioned earlier: meta slicing is performed on an event dependence graph and generates a reduced log; dynamic slicing is performed on the statement level dynamic dependence graph that is constructed during a fast forwarded replay.

3. THE EFF SYSTEM

As we mentioned earlier, tracing can handle an execution of up to a few seconds, whereas checkpoints are usually created with an interval of minutes. The ultimate goal of EFF is to fill the gap between tracing and checkpointing such that dynamic slicing can be applied. We have discussed how EFF fast forwards an execution from *the beginning* by replaying a reduced log. However, there is nothing fundamental that prevents EFF from being applied to executions resumed from checkpoints. Therefore, in this section we are going to describe how EFF, checkpointing, and tracing are integrated together. The composed system can be used to debug long running programs.

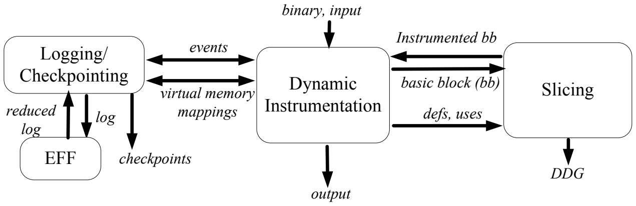


Figure 6: System infrastructure.

The system is presented in Figure 6. It consists of four components: *dynamic instrumentation* component, whose primary duty is to provide the infrastructure for tracing; *logging/checkpointing* component; *slicing* component; and *EFF* component. The system works as follows. In the original run, the slicing component is deactivated to reduce runtime overhead. The dynamic instrumentation engine traps each system call and forwards it to the corresponding handler in the logging model, which in turn logs the event. Checkpoints are created occasionally until a crash happens. In a replayed run, a smaller log file produced by EFF is supplied to drive the replay; in the mean time, the slicing component is turned on to trace the exercised data dependences till the crash point. The constructed dependence graph is studied to identify the root cause of a failure.

Dynamic Instrumentation. The dynamic instrumentation engine is adapted from *valgrind* [18], which takes a x86 binary and before executing any new (never instrumented) basic blocks it calls the instrumentation function provided. The instrumentation function instruments the given basic block and returns a new basic block to valgrind. Valgrind executes the instrumented basic block instead of the original one. The instrumented basic block is copied to a new code space and thus it can be reused without calling the instrumenter again. The instrumentation is dynamic in the sense that the user can enforce the expiration of any instrumented basic block such that the original basic block has to be instrumented again (i.e., instrumentation can be turned on and off as desired). In our case, we can easily turn on/off the slicing component for the sake of performance or for certain part of the code, e.g. library code.

Logging/Checkpointing. Our logging component is modified from *jockey* [17], which is an industry-strength checkpointing/replaying library executed in the application's space. Compared to the checkpointing techniques executed in the kernel space, jockey has superior usability. Setting LD_LIBRARY_PRELOAD=libjockey.so is the only command required to activate jockey. Once loaded in, jockey calls an initialization method before the application gains control. During the initialization, jockey scans through all the binaries including the libraries loaded by the application, looking for any system call sites. Those system calls are redirected to jockey in order to log the corresponding events or, during the replay, retrieve the events from the log file without actually passing them on to the OS. Checkpoints can be created by setting a timer, such that the application is not even aware of the existence of jockey, or by making a library call to jockey inside the application. In the latter case, the application has to include jockey's header files and be explicitly linked with the library. On receiving a checkpointing request, jockey obtains the layout of the application's virtual space, which is the jockey's space as well, by parsing */proc/self/maps*. A checkpoint is created by dumping all the virtual memory segments that do not belong to jockey.

Slicing. The slicing component is inherited from our prior work [23]. The main difference is that we do not trace control dependence in this system because according to our study [23] tracing only data dependence is powerful enough to capture the root causes of memory bugs, which are the ones usually reported for long running programs. Another difference is that we augment the component such that it stops at the execution points where illegal memory accesses occur, for example a write to an unallocated address. These points are usually earlier than the actual crash points.

EFF. The *EFF* component implements the technique described in the previous section. It takes an event log file dumped by the logging model and then computes the meta slice for a given set of events. The meta slice is used to prune the event log. The resulting smaller event log is used to drive the replayed execution. The computation of meta slice requires a static dependence graph, which is created by profiling the executed dependences in a few profiling runs due to the lack of an implementation of points-to analysis for x86 binaries.

One of the key challenges is to integrate logging/checkpointing model into the dynamic instrumentation engine. The integration is very meaningful because of the following reasons. Dynamic instrumentation is becoming more and more widely used in recent years. Not only is it attractive for the purpose of adaptive profiling/tracing, but also performance improvement can be achieved by executing a regular application on a dynamic instrumentation engine. Binary translation, a very promising technique that is derived from dynamic instrumentation, can virtually execute an architecture specific binary on a different architecture. Logging/checkpointing, on the other hand, has already been very popular for fault tolerance, debugging, etc. We believe logging/checkpointing should become a standard functionality of a dynamic instrumentation infrastructure. Therefore, the issues we are addressing here may be general to the integration of tools with similar functionality. The first issue is the separation of the virtual space. Both valgrind and jockey are residents in the application's space. They both assume total control over the virtual space such that they reserve certain address space for their own purposes. The reservations conflict each other. For instance, jockey reserves 0x7200000-0x7800000 for its heap, the mapping of the log file, etc. The same address space is also reserved by valgrind for tracing. Our solution is to make them aware of the existence of each other by separating the application's address space into two parts – the valgrind's space and the jockey's space. The application is actually executed within the valgrind's space. The second issue is about adjusting the system call trapping mechanism in jockey. Jockey traps system calls by directly overwriting the application's code. As a result, valgrind traces into jockey and tries to instrument the jockey code, which is undesirable. Our solution is to avoid any direct interactions between the application code and the jockey code. Jockey can only interface with valgrind. More precisely, we use valgrind to trap system calls and then call the jockey event handlers inside valgrind. The third issue is to discretely checkpoint the execution. A naive solution only checkpoints the application's program status. The reality is that the application's program status is so mixed up with the valgrind's status that valgrind fails to resume from checkpoints during replay if only the application's status has been checkpointed. Our solution is to treat the valgrind's status as part of the application's status such that it is checkpointed as well. Some of the valgrind's status should be excluded such as the valgrind's log file descriptor, which should be reopened at the beginning of a replay. There are some other minor issues in order to make both valgrind and jockey run correctly such as some of the valgrind's sanity checks have to be turned off.

4. EXPERIMENTATION

We need to address a few issues in order to carry out experiments. The first issue is that of selecting benchmarks to use in the experiments. The programs we select should be able to run for a long time. We looked at the set of bugs studied in [7, 13, 12] and picked the programs that can execute for a long time. Table 2 presents the set of programs we selected. Most of them are user interactive programs. We ignored *apache* since *apache* creates multiple processes while our logging model can handle only one process at the current stage. The second issue is that we need the input that can drive the execution for a long time and then crash. On the other hand, the execution should not be so long that it becomes too heavy a task for us to collect the data. Unfortunately, the input coming with the selected bugs usually leads to very short executions. Given the fact that most benchmarks are interactive, we constructed a long input by first performing a sequence of user actions and then applying the failure inducing input –the input comes with the benchmarks. For example in *mutt*, we took the following actions: (i) open an email account; (ii) go through all the emails one by one, the total is about six hundreds; (iii) try to switch to an invalid folder; repeat steps (ii) and (iii) two more times; provide the failure inducing input and hence crash the program. We collected the user time as the performance indicator since the real time may significantly differ each time depending on the user's behavior.

Table 2: Description of the benchmarks

Benchmark	Description	LOC	Bug Type
bc-1.06	interactive calculator	14.4K	heap overflow
mc-4.5.55	file manager	86.2K	stack overflow
mutt-1.4.2.1i	email client	453.6K	heap overflow
pine-4.44	email client	211.9K	stack overflow
pine-4.44	email client	211.9K	heap overflow
squid-2.3	web proxy cache server	93.5K	heap overflow

We investigated four execution scenarios: *orig.* denotes the original execution; *traced* denotes the original execution plus the de-

Table 3: Performance comparison of different execution scenarios.

Benchmark	Orig. (sec.)	Traced (sec.)	Traced/Orig.	Logged (sec.)	Logged/Orig.	EFF (sec.)	Traced/EFF
bc-1.06	13.6	2040.4	150.6	16.2	1.19	0.05	40808.8
mc-4.5.55	10.28	417.8	40.64	13.47	1.31	0.05	8356
mutt-1.4.2.1i	19.7	3237.7	164.5	26.1	1.32	0.06	53960.8
pine-4.44(stack)	14.4	2088.4	145.1	36.8	2.55	0.12	17403.6
pine-4.44(heap)	13.9	2102.2	151.5	34.4	2.47	0.20	10510.9
squid-2.3	14.6	1131.6	77.3	25.6	1.75	0.17	6656.4

pendence tracing; *logged* represents the original execution plus logging; *EFF* represents the fast forwarded execution plus the dependence tracing. In the logged run, an event log is created. The EFF technique is applied to reduce the log. The statement instance we want to replay is where the crash happened. The EFF technique is able to reproduce the crash in a much shorter execution. Due to the complexity of our system, our implementation is not sound at the current stage. Some times we have to hard code a few event dependences, otherwise the reduced log is not valid to drive the replay which is manifested as an event missing when it is expected or the presence of an extra event.

Table 3 compares the performance under the four scenarios. We can see the original runs, which were terminated by crashes, consume user time ranging from 10.2 to 19.7 seconds, which corresponds to the real time of a few minutes. They are not long by simply looking at the raw numbers, but they well exceed the capability of our dependence tracing technique. We can easily extend the executions by repeating the user actions. The side effect is the increased difficulty of collecting the execution time in the *traced* scenario. Note that even though checkpointing is supported in our system, the original execution does not last long enough to trigger it. Fortunately, it does not affect the evaluations of the EFF technique and the effectiveness of dynamic slicing on long running programs. From Table 3, we have the following observations.

- Dependence tracing introduces 40.64 to 164.5 times slow down. A programmer may accept it for a short run but highly unlikely for a long run.
- The slow down factor of logging ranges from 1.19 to 2.55, which is significantly smaller than the tracing slow down factor. For user interactive programs, the overhead is not human noticeable.
- EFF can greatly shorten an execution such that the overhead of dependence tracing becomes acceptable.

Table 4: Comparison of the event logs.

Benchmark	# of events in Orig.	# of events in EFF	Orig./EFF
bc-1.06	340509	7	48644.0
mc-4.5.55	322172	16020	20.1
mutt-1.4.2.1i	262559	489	536.9
pine-4.44	7365830	3028	2432.6
pine-4.44	8707316	27279	319.2
squid-2.3	1620988	795	2038.9

Table 4 compares the numbers of events before and after event reduction. We can see the reduction factor ranges from 20.1 to 48644.0, which well explain why the fast forwarded executions become so short. Table 5 presents the numbers of the exercised data dependences in the original and the fast forwarded executions. We want to point out that these numbers are collected after the intra-basic-block optimization [22] which eliminates considerable redundant dependences. We can tell that the numbers for the fast forwarded executions are much smaller. The constructed dependence graphs can be stored even without further compression [22].

Table 5: Comparison of the dependence graphs.

Benchmark	# of dep. in Orig.	# of dep. in EFF	Orig./EFF
bc-1.06	2.18×10^{10}	4.9×10^{5}	44489.8
mc-4.5.55	0.69×10^{10}	9.6×10^{7}	71.8
mutt-1.4.2.1i	4.86×10^{10}	4.21×10^{7}	1154.4
pine-4.44	1.95×10^{10}	2.68×10^{7}	727.6
pine-4.44	2.78×10^{10}	1.55×10^{8}	179.4
squid-2.3	1.1×10^{10}	1.93×10^{6}	5699.5

5. DYNAMIC SLICING ON A SET OF LONG RUNNING BUGS

The performance of a set of long running bugs has been studied in the last section. As the original motivation, dynamic slicing is applied and evaluated to show the effectiveness.

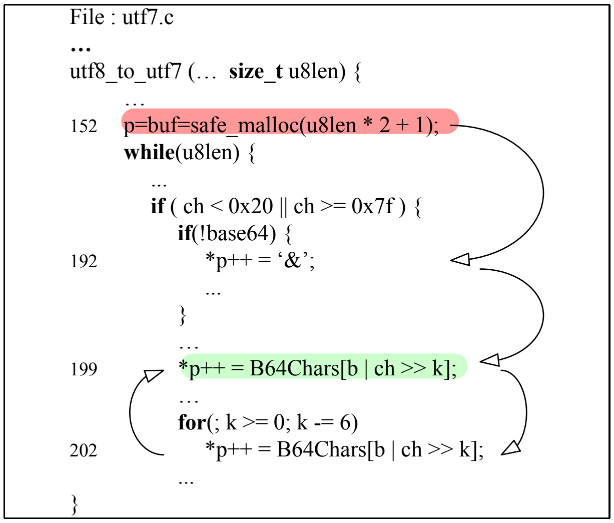
5.1 Mutt

Mutt [27] is a text based mail user agent (MUA) for Unix based Operating Systems. It has many features including customizability, POP3 and IMAP support, and ability to handle multiple mailbox formats. According to [28], mutt version 1.4 has a known memory bug which is as follows. The Mutt Mail User Agent (MUA) has support for accessing remote mailboxes through the IMAP protocol. When mutt has to convert the name of the folder from its internal UTF-8 representation to UTF-7 it calls the function *utf8_to_utf7* in module *imap/utf7.c*. When this function does the conversion, it miscalculates the length of the output string. To conduct our experiment, after Mutt is executed for a long time, we supply a UTF-8 folder name that contains some special characters. The heap buffer is overflowed and a segmentation fault is flagged. We reduce the event log using EFF and then replay the execution with the new log. Dynamic slicing is activated in the replayed execution. Figure 7 shows the computed dynamic slice.

As we mentioned earlier, our slicing component also monitors for any attempt for illegal memory access. After detecting a write to a memory region not allocated at line number 199, we now inspect the data slice to find the root cause. We find that the last instance of line 199 is data dependent on line 202 and vice-versa through variable 'p'. The arrows indicate the data dependence. The data dependence chain in the slice leads us to the first instance of line 199 which is data dependent on line 192 and this in turn is data dependent on line 152, which is the root cause of the bug as there is an error in calculating the buffer length at this point. We needed to inspect just 8 static statements before getting to the root cause, and the dependence chain provides a very clear explanation on the cause effect relations.

5.2 Pine

Pine [31] is a popularly used application for reading, sending and managing email messages and is distributed with the Linux operating system. Pine version 4.44 has two buffer overflow errors. One is a stack overflow and the other is a heap overflow. We look at both errors in the following subsections.



```
File : utf7.c
...
utf8_to_utf7 (... size_t u8len) {
    ...
152 p=buf=safe_malloc(u8len * 2 + 1);
    while(u8len) {
      ...
      if ( ch < 0x20 || ch >= 0x7f ) {
        if(!base64) {
192       *p++ = '&';
          ...
        }
        ...
199     *p++ = B64Chars[b | ch >> k];
        ...
        for(; k >= 0; k -= 6)
202       *p++ = B64Chars[b | ch >> k];
        ...
}
```

Figure 7: Mut 1.4.2.1i

5.2.1 *Pine Stack Overflow*

According to [32] pine has a stack overflow error. Pine calls an error prone API when it accesses mailboxes. By asking pine to handle a mailbox that has some special characters this bug can be triggered causing pine to crash.

We are able to capture the root cause of the bug again using dynamic slicing. Our tracking infrastructure reports an illegal memory access at line number 589 of file *mail.c*, where the statement is "for (...;(c=*t++)!='"';) {". We look at the slice at this point and find that there is a loop carry self dependence. This line is also the root cause of the BUG as variable 't', which is the pointer to a string, is incremented beyond its allocated region (on stack) if the provided string does not have the end quote. We needed to inspect 3 static statements to nail the root cause.

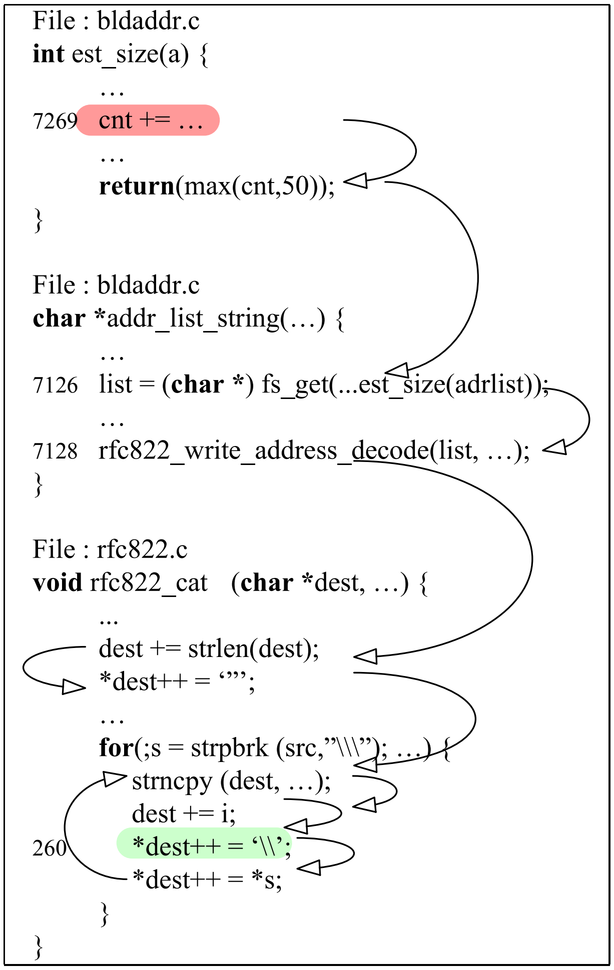
5.2.2 *Pine Heap Overflow*

According to [33] pine has a bug that when triggered can overflow the heap memory causing a potential crash. This can occur when pine processes the "From" field of email headers. Certain special characters in the header can cause the bug. Figure 8 shows the code where the bug is present.

There is an illegal heap access detected by our infrastructure at line number 260 in file *rfc822.c*. However, the root cause of the bug is at line number 7269 of file *bldaddr.c*. The buffer $dest$ in *rfc822_cat* is allocated in *addr_list_string*. The size of the allocation is miscalculated in *est_size* because it does not consider special characters. The figure shows the dependences that we tracked to get to the root cause from the error point. This is an example where the root cause and the symptom are in different functions. We had to examine 10 static code statements to get to the root cause.

5.3 Midnight Commander

Midnight Commander (mc) [29] is an open source file manager for free operating systems. It has high degree of portability and can be compiled and run on a number of operating system including Linux. We used mc version 4.5.55 for our experiment. This version has a known buffer overflow error. According to [30], the bug is triggered when midnight commander is used to process symbolic links in *tgz* archives. Absolute symbolic links in the archives are translated into links relative to the start of the *tgz* file. The buffer

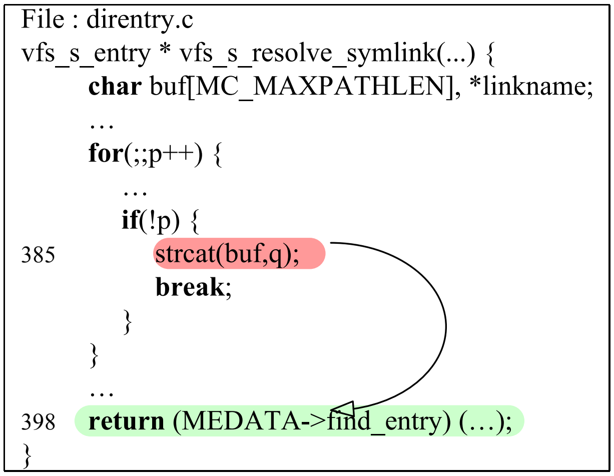


```
File : bldaddr.c
int est_size(a) {
      ...
7269  cnt += ...
      ...
      return(max(cnt,50));
}

File : bldaddr.c
char *addr_list_string(...) {
      ...
7126  list = (char *) fs_get(...est_size(adrlist));
      ...
7128  rfc822_write_address_decode(list, ...);
}

File : rfc822.c
void rfc822_cat   (char *dest, ...) {
      ...
      dest += strlen(dest);
      *dest++ = '"';
      ...
      for(;s = strpbrk (src,"\\\""); ...) {
        strncpy (dest, ...);
        dest += i;
260     *dest++ = '\\';
        *dest++ = *s;
      }
}
```

Figure 8: Pine 4.44 heap overflow.



```
File : direntry.c
vfs_s_entry * vfs_s_resolve_symlink(...) {
      char buf[MC_MAXPATHLEN], *linkname;
      ...
      for(;;p++) {
        ...
        if(!p) {
385       strcat(buf,q);
          break;
        }
      }
      ...
398   return (MEDATA->find_entry) (...);
}
```

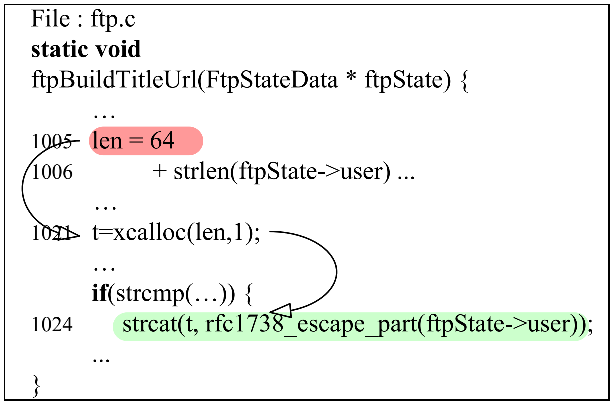
Figure 9: Mc 4.5.55

that is used to form the relative link is never initialized and hence can be overflowed inside the *strcat* procedure. Figure 9 shows the code corresponding to the bug.

We use our infrastructure to determine the root cause of the bug. A segment fault occurs at line number 398. Now, when we look at the slice at this point we find an abnormal data dependence between line 398 and line 385. We conclude that a stack buffer overflow happened at line 385, which is the root cause of the bug, such that it corrupted one of the variables used at line 398. We just needed to inspect 2 static statements to get to the error.

5.4 Squid

Squid [34, 7] is a fully featured web proxy cache that supports proxying and caching of HTTP, FTP and other URLs. It is designed to run on Unix based systems. We use squid version 2.3 for our experiment. It has a known heap buffer overflow error. When an input request contains some special characters, squid miscalculates the length of the heap buffer that is used to hold the request. As a result, the buffer is overflowed and then the server crashes. [35] explains it in more details. Figure 10 shows the portion of the code that contains the bug.



```
File : ftp.c
static void
ftpBuildTitleUrl(FtpStateData * ftpState) {
        …
1005    len = 64
1006            + strlen(ftpState->user) ...
        …
1021    t=xcalloc(len,1);
        …
        if(strcmp(…)) {
1024        strcat(t, rfc1738_escape_part(ftpState->user));
        ...
}
```

Figure 10: Squid 2.3

On running squid using our infrastructure we find that there is a heap buffer overflow at line number 1024. Inspecting the slice at this point leads us to the root cause of the bug at line number 1005, at which the extra padding space of size 64 is not enough to accommodate the special characters. We had gone through 5 static statements before we reached the root cause.

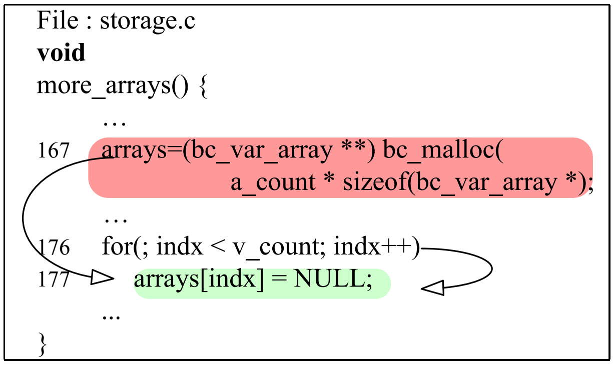
5.5 bc

Bc [26] is a numeric processing language that supports arbitrary precision numbers. It is generally distributed along with the Linux operating system and is a part of the GNU project. We used bc-1.06 for our experiment. This version has a known heap overflow error.

In [6, 7] the bug that is triggered in bc is described. A certain heap buffer is not declared wide enough and overflows. The code corresponding to the error is shown in Figure 11. The heap array *arrays* declared at line number 167 is overflowed.

Our tracking infrastructure detects a heap memory violation at line number 177. Looking at the slice at this point we see that the root cause of the bug is at line number 167. This is because *a_count* entries have been declared but *v_count* entries are accessed. We needed to inspect just these 3 statements to find the root cause.

From the studies we find that these bugs are not as mysterious as they appear, under the micro-inspection of dynamic slicing. They usually require examining a few static statements before the root cause is located. Two conclusions can be drawn: dynamic slicing is very effective to handle memory type of bugs even in the long running programs examined; the real challenge is to isolate the part of the execution that is relevant to the error and hence dynamic slicing can be applied. The EFF technique is designed for the purpose. According to our experience in [23, 3, 24], most non-memory bugs still have very good locality even though not as apparent as memory bugs. We firmly believe EFF plus dynamic slicing will still be highly effective for non-memory bugs in long running programs. Unfortunately, most bugs that are reported and studied for those programs are memory bugs. We plan to mine some software repositories of long running programs to get more interesting non-memory bugs in the future.



```
File : storage.c
void
more_arrays() {
      …
167   arrays=(bc_var_array **) bc_malloc(
                a_count * sizeof(bc_var_array *);
      …
176   for(; indx < v_count; indx++)
177       arrays[indx] = NULL;
      ...
}
```

Figure 11: Bc-1.06

6. RELATED WORK

The work that is very closely related to ours is the delta debugging technique [21]. Delta debugging is similar to our work in terms of their ability to reduce a failed execution without losing the capability of reproducing the failure. Delta debugging is essentially a systematic search algorithm which can minimize failure inducing input and, if given both a successful run and a failed run, isolate the minimal failure inducing input difference between these two runs. The basic idea is to use a binary search alike algorithm to generate different combinations of input and then re-execute the program with these input to see if the failure can be reproduced. In [21], Zeller et al. also treated the interactions between an application and the user as input and applied delta debugging to identify the minimal sequence of failure inducing interactions. In [10], Orso et al. capture the interactions between different software components such as method calls, and then apply delta debugging to identify the minimal sequence of failure inducing interactions before replay. In [2], Choi et al. applies delta debugging to isolate the failure inducing thread schedule difference.

The difference between delta debugging and our work is that delta debugging is a black box technique while our technique adopts a white box strategy which tries to reduce an execution by inspecting dependences between events. If the execution is short, applying delta debugging to the captured events may perform better since reexecuting the program takes very little time. However, if the execution is long and thus the volume of events is high, a large number of reexecutions, which is the inevitable result of the search used by delta debugging, is not desirable. On the other hand, we believe delta debugging can be an addition to our technique. For example, we found it is hard to disclose some data dependences between events without being conservative. However, being conservative implies that we often fail to remove certain irrelevant events. Under such circumstances, a black box strategy such as delta debugging can be employed to systematically determine if these dependences should be considered during reduction. Finally, the goal of our technique is not merely execution reduction, but to make heavyweight tracing technique as dynamic slicing feasible.

Another stream of related work includes [9, 16]. In these works, all the interactions between a software unit and other external components are captured as events. As a result, the software unit can be replayed based the log without the need to execute the remaining part of the software. These techniques are useful in unit testing while our technique is more general. Furthermore, we also focus on locating errors through dynamic slicing.

7. CONCLUSIONS

We have enabled dynamic slicing on a set of long running programs by developing a novel execution fast forwarding technique. Fast forwarding can be achieved by driving the replay with a reduced event log file. Given a desired execution region, a large portion of the events are not relevant to replaying it. Meta slicing is designed to eliminate this redundancy in the log file. With the execution fast forwarding technique, the replayed execution becomes substantially shorter and yet the wanted execution region is precisely reproduced. The reduction factors of the sizes of dynamic dependence graphs range from 72 to 44490. As a result, dynamic slicing can be practically applied to isolate the cause effect chain leading to the failure. Our studies show that most of the reported memory bugs for long running programs are trivial to locate with dynamic slicing once the execution has been shortened to an affordable level.

Acknowledgements

This work is supported by grants from Microsoft, IBM, and NSF grants CNS-0614707, CCF-0541382, and CCF-0324969 to the University of Arizona.

8. REFERENCES

[1] S. Bhansali, W-K. Chen, S. de Jong, A. Edwards, R. Murray, M. Drinic, D. Mihocka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," *Virtual Execution Environments Conference*, Ottawa, Canada, June 2006.

[2] J. Choi and A. Zeller, "Isolating Failure-Inducing Thread Schedules", *Proceedings of the International Symposium on Software Testing and Analysis*, Rome, Italy, July 2002.

[3] N. Gupta, H. He, X. Zhang, and R.Gupta, "Locating Faulty Code Using Failure-Inducing Chops," *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 263-272, Long Beach, California, Nov. 2005.

[4] T. Gyimothy, A. Beszedes, I. Forgacs, "An efficient relevant slicing method for debugging," *7th European Software Engineering Conference/ 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Toulouse, France, 1999.

[5] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, Vol. 29, No. 3, pages 155-163, 1988.

[6] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug Isolation via Remote Program Sampling," *SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[7] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: a benchmark for evaluating bug detection tools", *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[8] R.H.B. Netzer and M.H. Weaver, "Optimal Tracing and Incremental Reexecution for Debugging Long-Running Programs", *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, USA, pages 313-325, June 1994.

[9] A. Orso, and B. Kennedy, "Selective capture and replay of program executions", *In Proceedings of the Third international Workshop on Dynamic Analysis*, St. Louis, Missouri, May 17 - 17, 2005.

[10] A. Orso, S. Joshi, M. Burger, and A. Zeller, "Locating Causes of Program Failures", *Proceedings of the 2006 International Workshop on Dynamic Analysis*, Shanghai, China, May 2006.

[11] D.Z. Pan and M.A. Linton, "Supporting reverse execution of parallel programs," *ACM workshop on parallel and distributed debugging*, Madison, WI, USA, May 1988.

[12] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies - a safe method to survive software failures", *the 20th ACM Symposium on Operating Systems Principles* Brighton, UK, pages 235-248, Oct. 2005

[13] M.C. Rinard, C. Cadar, D. Dumitran, D.M. Roy, T. Leu, and W.S. Beebee, "Enhancing Server Availability and Security Through Failure-Oblivious Computing", *the Sixth Symposium on Operating System Design and Implementation* San Francisco, California, pages 303-316, 2004

[14] M. Ronsse, K. De Bosschere, M. Christiaens, J.C. de Kergommeaux, and D. Kranzlmller, "Record/replay for nondeterministic program executions", *Communication of the ACM* 46(9), pages 62-67, 2003

[15] M. Ronsse, K. De Bosschere, and J.C. de Kergommeaux, "Execution replay and debugging", *Fourth Workshop on Automated and Analysis-Driven Debugging*, Munich, Germany, August 2000.

[16] D. Saff, S. Artzi, J.H. Perkins, and M.D. Ernst "Automatic test factoring for java", *In Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering*, Long Beach, CA, USA, November 07 - 11, 2005.

[17] Y. Saito, "Jockey: a user-space library for record-replay debugging", *Sixth International Symposium on Automated and Analysis-Driven Debugging*, Monterey, California, September 2005.

[18] J. Seward et al. "Valgrind: A GPL'd system for debugging and profiling x86-linux programs", *http://valgrind.ked.org/*, 2004.

[19] S.M. Srinivasan, S. Kandula, C.R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging", *USENIX Annual Technical Conference*, Boston, MA, USA, June 1994.

[20] L.D. Wittie. "Debugging distributed C programs by real time replay," *ACM workshop on parallel and distributed debugging*, pages 57-67, Madison, WI, USA, May 1988.

[21] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input", *IEEE Transactions on Software Engineering 28(2)* , February 2002, pp. 183-200.

[22] X. Zhang and R. Gupta, "Whole Execution Traces," *IEEE/ACM 37th International Symposium on Microarchitecture*, pages 105-116, 2004.

[23] X. Zhang, N. Gupta and R. Gupta, "A Study of Effectiveness of Dynamic Slicing in Locating Real Faults," *Empirical Software Engineering* Journal, August 2006.

[24] X. Zhang, N. Gupta, and R. Gupta "Locating Faults Through Automated Predicate Switching," *IEEE/ACM International Conference on Software Engineering*, Shanghai, China, May 2006

[25] X. Zhang, N. Gupta, and R. Gupta "Pruning Dynamic Slices With Confidence," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Ottawa, Canada, June 2006

[26] GNU bc. http://www.gnu.org/software/bc

[27] Mutt Website. www.mutt.org

[28] Mutt Buffer Overflow. http://www.securiteam.com/unixfocus/5FP0T0U9FU.html

[29] Midnight Commander. www.ibiblio.org/mc

[30] Midnight Commander exploit. www.securityfocus.com/bid/8658

[31] Pine Website. www.washington.edu/pine/

[32] Pine Stack Buffer Overflow Error. http://www.xatrix.org/advisory.php?s=7408

[33] Pine Heap Buffer Overflow Error. http://www.securityfocus.com/bid/6120

[34] Squid Website. http://www.squid-cache.org/

[35] Squid Buffer Overflow. http://www.securiteam.com/unixfocus/5BP0P2A6AY.html