Application Level Undo & Recovery: Applied to the Pencil Application on Linux

Rafat Rashid (996096111), Bozhidar Lenchov (995959431), Kush Dua (996081957)

December 12, 2012

Abstract

Pencil is an open source, multi-platform QT-based drawing and animation solution. Software bugs in the program can cause the application to exhibit anomalous behaviour or degrade in performance. This can cause the program to hang or crash which can ultimately lead to the loss of user data. We introduce undo and recovery functionality into Pencil that will help the user recover their work by replaying logged operations onto periodically recorded snapshots of the canvas. Evaluation of the operation logging solution concluded acceptable real-time costs in the range of 20-30 ms for performing per mouse movement in an operation, 7-10 kB storage requirement per draw event per second and 4-6 ms of recovery per draw action on application startup following a crash. Similar snapshot collection overheads of 2-3 kB storage and 8-10 ms processing for each draw event were already present in the system. Although some of these numbers may seem prohibitively high, we feel the positive aspect of accurately and completely recovering user work presents a much greater benefit for the use of this recovery feature.

1 Introduction

Pencil is an open-source 2D animation and drawing application software for Mac OS X, Windows and Linux [17]. The UI is developed using QT and the engine is written in C++. The app supports external plugins, bitmap and vector graphics, and is able to export Flash animation sequences. In this report, we present the addition of undo and recovery functionality into Pencil in the face of crashes due to software bugs or other unforeseen events.

We begin with a discussion of the motivation for our work and present the objectives, constraints imposed to make our task feasible and our contributions. In Section 2, we provide an overview of the Pencil application and its modules. We also briefly describe which components were modified and what functionalities were added. Section 3 presents a detailed account of the work completed. In Section 4, we discuss our evaluation methodology and Section 5 presents our results. Section 6 presents related work in application snapshot and undo and recovery mechanisms. Section 7 provides a discussion of future work in improving the dependability of Pencil, including how to extend the work presented here. Finally Section 8 concludes the report.

1.1 Motivation

In practice, it is very difficult to achieve bug free code [14]. Software bugs can result from memory leaks, out of bounds memory access and integer overflows to name a few. These can cause corruption of Pencil's documents or performance degradation. However, software bugs are not the only sources of software failures. Jim Gray's analysis in 1985 found that 42% of failures were due to configuration and operator errors, 25% were due to software bugs, 18% from hardware failures and 14% were attributed to power failures and the environment [9]. This remains an open problem in software systems to this day. Eliminating operator and configuration errors for example is extremely difficult if not impossible [1][2].

Heisenbugs such as race conditions, resource leaks and environment dependent bugs are extremely difficult to find, patch and recover from [4]. Further, according to Alan Wood's report, up to 80% of the bugs in production systems have no fix available at the time of the failure [26]. This is exacerbated with the existence of external factors such as viruses, worms and trojans that can take the form of an imported plugin or image with the sole intent of exploiting a system or an application.

Crash inducing buffer overflows and out of bounds memory accesses are common forms of vulnerabilities in user applications [5]. They can result from poor input validation and programmatic errors, accounting for 13% of vulnerabilities according to Veracode in 2011 [27]. Pencil provides plugin support and importing of images, sounds and configurations files. This can be exploited in many different ways and accounts for a very large portion of possible vulnerabilities [27].

We can strengthen software against buffer overflow and out of bounds memory access with detection [20][21]

and recovery [22]. Exterminator [15] uses address space randomization to detect out of bounds writes into the heap and accesses via dangling references. Software error detectors and recovery mechanisms for numerous forms of other software bugs are discussed in ClearView [18]. ClearView automatically finds and patches errors in applications without requiring checkpoints and restart/rollback. This is useful for systems that require high availability but can also result in real-time performance degradation for long running applications.

For user applications like Pencil, we argue performance, protecting against data corruption and running in a clean state is more crucial than reducing interruptions in a user's workflow. A more suitable option is therefore a checkpoint and recovery mechanism similar to the work done in Undo for Operators [3].

1.2 Our Contributions

From our discussion of bugs, failures, their origins and of Pencil's vulnerability against them, one can deduce the benefits of a recovery solution that can be oblivious to them. We improve the dependability of Pencil by:

1. Taking snapshots of Pencil state (e.g. canvas contents) at regular intervals between user save events.

2. Logging operations (such as paint strokes by the pen, pencil and eraser tools) necessary for replaying user actions between snapshots to recover user progress on application restart following a crash.

Our undo and replay mechanism is only possible if the state of Pencil does not change outside of the logged operations and snapshots. This is typical of existing log-based rollback and recovery systems [7][11]. On application restart, if the recovery process fails due to the faulty operation being executed again, the user can simply try again (e.g. specify less operations to replay) until it is successful (refer to Section 7.1). This is possible because we do not make any modifications to the snapshots or the logged files during recovery. Our solution does not protect against application crashes or detect and repair software bugs automatically.

We focus on undo and recovery for the canvas contents in the bitmap and vector layer animation sequences and tool state. To limit the scope of our work, software bugs that cause propagation failures were not considered. For instance, a crash after user save and associated deletion of snapshots and log data event due to a bug at an earlier point of Pencil's execution is not supported. These failures can also result in logical operations being logged after the bug manifests, which can be hard to detect and resolve. One possible solution for this is delaying operation logging or snapshot removal. For example, deleting snapshot contents after another five or more draw events have completed can be done to ensure no delayed crash occurs before backup replay data is removed.

2 Overview of Pencil

Pencil is composed of three main modules: 1) Interface 2) Structure and 3) Graphics. The Interface module contains all the classes related to the user interface. This includes the windows, menus, panels and the scribble area (or canvas). The Structure module contains all the classes that represent a Pencil document. This includes maintaining the bitmap and vector layers of the document object and saving/loading the document to/from disk. The Graphics module contains all the classes needed to process bitmap and vector graphics. Figure 1 illustrates a high-level schematic of the Pencil application and how the different modules interact with each other.

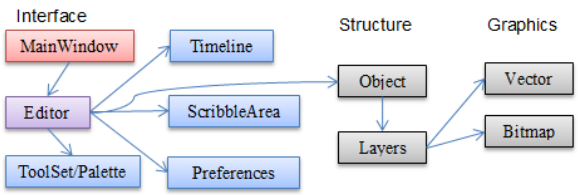

Figure 1: Structural Diagram of Pencil

The Timeline allows the user to navigate and manipulate animation (or vector) frames. The ScribbleArea allows the user to draw on the canvas using QT UI events. The ToolSet and Palette manages the drawing tools and colour palette respectively. The Preferences is a separate window that the user opens to manipulate application settings (including canvas snapshot frequency). The Editor manages all these components and orchestrates the flow of events and data between them. All of our code changes are limited to these components in the Interface module. The specific changes are discussed in Section 2.1 and 3 of this report.

2.1 Overview of Our Changes

For logging user operations, we focus on monitoring changes in the ToolSet, Palette and the ScribbleArea. The methods mousePressEvent(), mouseMoveEvent() and mouseReleaseEvent() serve as entry points. To update which tool is selected in the ToolSet, the method associated with the tool clicked (such as pencilOn(), penOn() and eraserOn()) is called. When the user clicks and drags the mouse to draw on the canvas with the pencil, pen, eraser and colouring tools, the drawLineTo() method is called. If the polyLine tool is selected, drawPolyline() method is called instead.

From the above description, we make the observation that Pencil has well defined semantics for user operations and that such operations are serializable. This is noted by the Undo for Operators paper as an important facility for logging and replay. The paper describes a Rewind, Repair and Replay mechanism which was implemented for an email store to recover from operator (user) errors [3]. These errors can result from bad inputs to the program, including unexpected configuration settings, data and plugins. Pencil is also perceptible to all three of these external inputs. Based on Jim Gray's analysis, 42% of failures are due to configuration and operator errors [9].

Tackling operator and configuration errors has been limited by performance and explosion in causality tracking of thousands of outside parameters [1][2]. One approach to dealing with problems resulting from external inputs is to detect and filter them. An example is VSEF which filters out attacks on a specific vulnerability based on the vulnerable program's execution trace [13]. These techniques would benefit Pencil as it supports external plugins that can add any functionality and reads/alter the settings of a configuration file during runtime.

Similar to Undo for Operators, we intercept and log user operations via the methods listed above. As a user drags the mouse, the mouseMoveEvent will be generated numerous times. Unlike the paper which logs the interactions at the boundary before they enter the system, we log each sub-operation after it is applied to the canvas, since we want to ensure the user saw the successful operation being applied. Unlike a Transaction based recovery system [12], if there is a crash after an operation is applied to the canvas but prior to committing the log, we will lose this information needed for the replay phase. Further discussion of the disk commit granularity is presented in Section 3.3.

Like Undo for Operators, we physically Rewind by loading a snapshot of the Pencil document and Replay using the logged logical user operations. As the paper explains, a physical rewind avoids the risk of rolling back from non-deterministic or corrupted state. Similarly, a physical replay would only cause the bug to manifest again. We instead replay user intentions such as tool strokes on the canvas.

Unlike Undo for Operators, we have not implemented a repair phase to prevent the problem from recurring during replay. This means if Pencil crashes due to a deterministic software bug, such as dereferencing a null pointer, replaying the user operation will make the application crash again because the same code path will be exercised during recovery. Virtual machines such as VMware [10] provide the ability to log system operations, rewind and replay but not repair. These systems, including the ones presented in [7][11] would not operate if the underlying environment is modified.

Changing the execution environment to prevent a bug from recurring is not trivial. We propose a workaround which is discussed in Section 7.1. Rx is another example of a solution that dynamically changes the execution environment (inputs, scheduling of threads, padding of allocated memory, etc) based on the type of fault that occurred after a rollback to a previous checkpoint [19]. It then replays the application in the repaired environment to avoid running into the problem again. For proper logging and recovery functionality of Pencil, in this report we assume our code and the Pencil methods discussed above have no crash inducing or data corruption bugs. Instead, we assume the bugs are a result of external inputs. The repair step is left for future work.

3 Implementation Details

The undo and recovery development has been broken into several milestones: Snapshot facilities, Log User Operations, Replay and Recovery and Evaluation. These features were implemented and evaluated on the Linux kernel using Qt4 libraries. The code modifications (including evaluation code) was not environment specific. We foresee cross-platform support with minimal programming effort. However, validating this is out of scope for this report. Each of the recovery milestones and the associated code changes are discussed in more detail next.

3.1 Snapshot Feature

Pencil allows the user to save the canvas drawing and animation frames as PNG images and an XML file that describes the layout of the frames. Similarly, we can load this Pencil document into the application for editing. We take advantage of this existing save/load functionality to take periodic snapshots of the users progress into a temporary location and log this event in a global XML file listing all snapshots and their associated files needed for replay. As part of the snapshot process, we also store operations involving tools such as the pen, pencil and eraser and their associated state (e.g. colour) into an operations XML log file used for replay later.

Taking diffs of successive snapshots was considered to reduce the storage overhead, however, we decided to take full snapshots in order to remove the dependence between snapshots due to data corruption bugs. Replay impact on storage requirements is evaluated in Section 5.3. On a successful user initiated save operation or normal program exit, these temporary snapshot and log files are removed.

The Preferences UI component was extended to let the user change the frequency of snapshots (number of drawing area actions before a snapshot is taken), the location of the temporary snapshot directory and ability to turn the

```
<!DOCTYPE PencilOperationsSaveLog>
<operations>
  <operation toolMode="0" layerType="1" currentLayer="0" currentFrame="1" opPosition="STARTING"/>
  <operation toolMode="0" lastPointY="-94" endPixelX="210" endPixelY="230" currentWidth="1" endPointX="-190"
currentColour="#000000" endPointY="-92" layerType="1" opPosition="DRAWING" antialiasing="1" lastPointX="-190"/>
  <operation toolMode="0" lastPointY="-92" endPixelX="210" endPixelY="231" currentWidth="1" endPointX="-190"
currentColour="#000000" endPointY="-91" layerType="1" opPosition="DRAWING" antialiasing="1" lastPointX="-190"/>
  <operation toolMode="0" lastPointY="-91" endPixelX="210" endPixelY="233" currentWidth="1" endPointX="-190"
currentColour="#000000" endPointY="-89" layerType="1" opPosition="DRAWING" antialiasing="1" lastPointX="-190"/>
  <operation toolMode="0" layerType="1" opPosition="ENDING"/>
...
  <operation toolMode="0" layerType="1" currentLayer="0" currentFrame="1" opPosition="STARTING"/>
  <operation toolMode="0" lastPointY="-65" endPixelX="236" endPixelY="262" currentWidth="1" endPointX="-164"
currentColour="#000000" endPointY="-60" layerType="1" opPosition="DRAWING" antialiasing="1" lastPointX="-167"/>
...
  <operation toolMode="0" lastPointY="-33" endPixelX="344" endPixelY="289" currentWidth="1" endPointX="-56"
currentColour="#000000" endPointY="-33" layerType="1" opPosition="DRAWING" antialiasing="1" lastPointX="-57"/>
  <operation toolMode="0" layerType="1" opPosition="ENDING"/>
</operations>
```

Figure 2: Sample Operations Log File

feature off and on. As mentioned, the Editor was modified to take periodic snapshots and remove all temporary files on a user initiated save operation or application close.

3.2 Log Operations

User operations performed on the drawing area and associated toolset and palette state are recorded in an XML state log file corresponding to the most recently taken snapshot through logical indexing. A central log file contains a listing of the saved snapshots and associated operation state XML files. A single global file handler is visible to each of the Interface modules components. We did not have to add logging code in the Structure and Graphics modules. As an example, a stroke on the canvas using the Pencil tool will produce a log similar to Figure 2.

The $<operations>$tag encapsulate a single user draw event (such as mouse press, drag and release). Each $<operation>$tag corresponds to a call to the drawLineTo method which gets called numerous times during this event, depending on the mouse drag movements. Since Pencil is single threaded, maintaining a single file descriptor is sufficient for an absolute logical ordering of events. Our solution does not care about timing of successive events so long as the logical ordering is maintained.

3.3 Replay and Recovery

Assuming there is a crash prior to a user save event, the existing snapshot files will remain. On application restart, this is detected and the automated recovery mechanism is initiated. First, the most recent snapshot is loaded onto the canvas using Pencil's facilities. The corresponding operations log file is then parsed to replay the recorded user operations (occurring after the selected snapshot and before the crash).

During testing, we found there was moderate visible slowdown when the log files was flushed to disk with every new $<operation>$tag. As a result, we chose to implement a trivial memory storage structure and flushed the file after a user-specified number of $<operation>$tags instead. We evaluate this approach in Section 5.2. However, by doing this, any operation not recorded in the file prior to the crash will be permanently lost. Finally, a crash during the recovery phase was not considered in this report but is an interesting area of investigation for the future.

4 Evaluation Methodology

Algorithm accuracy and performance of the snapshot, logging and recovery processes were evaluated to ensure that the program is still usable and does not have prohibitively high runtime processing and memory overhead. To evaluate correctness, we make sure the image reconstructed by the replay feature is the same as if it was saved by the user right before the instrumented crash took place.

For runtime performance, we evaluate the overheads introduced during runtime by measuring the time spent for taking snapshots and logging operations needed for undo-redo functionality. We instrument the code with crash inducing bugs to ensure the functional correctness of our features. We further evaluate the storage requirements of the snapshots and log files and measure the time it takes to recover user work after an application restart. Our evaluation methodology is described in more detail below.

4.1 Correctness

Verifying the correctness of the snapshot, logging and recovery algorithm is important for ensuring the user is able to accurately recover their work and not end up with a different canvas than the one prior to the crash. For testing purpose only, we turn off the functionality to remove snapshots and verify correctness as follows:

1. Image is saved at time 0 (e.g. *image0*).

2. User performs different operations from t0 -⟩ t1 and then saves the image in a new file (*image1*).

3. Immediately following the image save, the program unexpectedly closes (e.g. killed by user, although no bug has occurred).

4. Starting with *image0* and logged operations, the program would reconstruct the image (unaware of *image1* already being saved by the user).

5. Using a custom testing application and the QImage object equality operator, the two images are loaded into memory and compared against each other. Our solution is correct if these images match.

4.2 Runtime Performance

One of our major concerns was the runtime performance degradation introduced by our instrumentation. This is due to various overheads such as additional time used for logging every operation, writing to disk and, computational time spent on replaying operations from log files. Both of these I/O and computational time measurements are taken using standard QT timing measurement libraries. Since the undo and recovery changes are novel to the program, this information is gathered solely for displaying discrete values in the report and will not be compared to other application runs or alternatives.

4.3 Storage Requirements

Depending on the frequency of taking checkpoints and the jerkiness of mouse movements during a user draw event, the size of our logs can vary. Analysis of the correlation between these parameters is presented in the runtime and storage analysis in Section 5.3.

4.4 Introduction of Software Bugs

The main application of our solution is to recover user progress following a program crash. As such, code bugs and configuration/other bugs leading to degraded performance or crash were the main ones tested. For non-crash bugs leading to performance degradation, we assume program termination from the user, leading to a similar application halt as the crash bug. We found it is surprisingly easy to crash the program. For example, an extra parameter in the API signature of the plugin module causes Pencil to crash during import. One can think of more elaborate ways to cause faulty behaviour including stack smashing attack by overflowing a header parameter of the PNG file read in by Pencil's graphics component. However, due to time constraints, we left testing and evaluation of such scenarios to future work.

5 Evaluation Results

In this section, we analyze the results of our evaluation methodology discussed in the previous section. For our tests, the recovery process is automated, requiring no human input. Upon startup after a crash, Pencil loads the last snapshot into the canvas and applies subsequent logged user operations until the instrumented crash occurs. The crash occurs deterministically which allowed for an objective comparison of the overheads incurred by our features from one test case to another.

5.1 Correctness

We validated our snapshot, logging and recovery facilities by following the algorithm presented in Section 4.1. The topmost image in Figure 3 shows the restored snapshot (snap1) with replayed operations. The image in the middle shows the original snap1 snapshot. This is affirmation of the logged operations being applied on top of snap1 to generate the topmost image. Finally, the bottom part shows image1 which was saved before the simulated crash occurred. As shown in Figure 4, our comparator tool uses QImage's equality method native to QT to confirm the equality of the two images.

Figure 3: Top - replayed image; middle - snapshot; bottom - image before crash

```
grad@grad-desktop:~/Desktop/ece1724-pencil/QImageComparator2$ ./QImageComparator
 ../snapshots/image1.data/*png ../snapshots/image2.data/*png
The images are the same. =]
```

Figure 4: Replayed image is same as image before crash

5.2 Runtime Performance

A runtime and storage evaluation needs to take into account the number of operations saved in a single XML file as well as their duration. Prolonged draw distances as opposed to short strokes would result in a larger operations log since for each draw event (MousePress and MouseRelease) more log entries are saved. For the results presented in Table 1, the following parameters were changed:

- Number of draw events in each log file was varied between 5 and 10. This has no effect on the per operation runtime cost and linear effect in storage cost.
- Duration of user draw events. Each draw stroke is performed in real-time for 1 or 2 seconds. The duration was kept small to reduce the variance in the number of operations logged between test cases.
- Number of cached operations before they are flushed to disk are varied between 1, 2, 5 and 10.

User Events in Snapshot	5	10	5	10
User Event Duration	1	1	2	2
Operations tags cached	1	1	1	1
Operation Log Size (kB)	81	121.4	120.1	169.8
Avg Time Opening Log per Operation Tag (ms)	9.2	13.3	12.8	18.9
Avg Creating Operation Tag (ms)	0.1	0.1	0.1	0.1
Avg Time Closing File per Operation Tag (ms)	7.8	9.7	9.2	13.7
Draw Time per Operation Tag (ms)	0	0	0	0
Replay Time (ms)	38	60	56	63
Total Start up Time (ms)	299	320	320	319

Table 1: Runtime Performance Results

Table 1 shows an experiment where the number of operations and their draw duration is varied with the in-memory operation cache size kept constant. Varying the number of user events (such as a pencil stroke) does not directly affect the time needed by each operation tag to be processed. The per-tag processing time includes the file open/close and tag creation duration. The replay time upon startup however is linearly-related to the number of operations saved in the log file.

Table 2 below shows an experiment seeking to find correlation between the in-memory operations cache size and runtime. As expected, increasing the cache size (the number of operations saved in memory before flushing them to disk) decreases the average time spent closing the file. This is not reflected in opening of the log file, as most of the runtime costs are hidden by XML structure initialization and tag creations. Furthermore, runtime savings from keeping the operations log file open to hide some of the disk latency risks having open file descriptors during a crash and would also be hidden by the QT XML operations. Note that the cached operations will be lost if the program crashes before they are flushed out to disk but after they are applied to the canvas. A large cache size therefore may be intolerable depending on the use case.

User Events in Snapshot	10	10	10	10
User Event Duration	2	2	2	2
Operations tags cached	1	2	5	10
Operation Log Size (kB)	154.0	157.1	184.0	187.8
Avg Time Opening Log per Operation Tag (ms)	17.7	17.3	21.4	22.3
Avg Creating Operation Tag (ms)	0.1	0.1	0.1	0.1
Avg Time Closing File per Operation Tag (ms)	11.8	5.3	2.9	1.5
Draw Time per Operation Tag (ms)	0	0	0	0
Replay Time (ms)	40	60	62	64
Total Start up Time (ms)	280	304	309	310

Table 2: Varying Operations Cache Size

Both experiments above show user tolerable runtimes (20-30 ms per mouse movement for an operation, 4-6ms recovery per draw action), in addition to the 8-10ms overhead of saving snapshots for each user draw action, since the average users perception of delay occurs above the 100 ms point [16]. Future work could focus on decreasing the number of recorded operation tags associated with each user-initiated canvas event. This however has the tradeoff of losing an entire draw operation even if only a part of it was faulty. This could be due to a miscalculation resulting in out of bounds exception in one part of the stroke. By using finer operation logging like the current implementation, finer parts of a user event can be recovered. This is advantageous for long duration strokes.

5.3 Storage Requirements

As seen in the experiments presented in the runtime evaluation section, the operations log size has a linear correlation with the number of operations saved. On average, with the current fine grained operation tag implementation, we found each discrete operation to consume approximately 7-10kB/operation per second in disk storage of the snapshot and the logged files. This is in addition to existing Pencil snapshot functionality which required 2-3kB per draw event per second in our testing. By a similar argument as in the runtime evaluation section, logging less of the mouse events of the discrete draw operation could result in lower storage requirements but would decrease the recoverability of parts of the user event.

Associated with runtime, although the costs of removing snapshot and other associated recovery files is linearly related to the number of these files, it is less so compared to the size of the files. We think this is due to the removal operation operating on inodes on the OS level. A

fairly constant runtime cost of 15-20 ms was observed for removal of all snapshots and log files in the limited operational testing we performed.

5.4 Introduction of Software Bugs

Irrespective of the cause of the crash, the recovery process is the same. It is completely agnostic toward which bug caused the crash. Several types of bugs were emulated, all resulting in successful restoration of canvas state, similar to correctness testing. We introduced:

- Deterministic bugs (assertion faults at certain code execution paths either at each or after a certain number of runs, out of bounds exceptions, segfaults)
- Non-deterministic bugs (assertion failures using random number generation)
- Performance degradation causing program termination by emulation of misconfiguration impacts (e.g. inducing sleep in certain code paths)

For succinctness and avoiding repetition, further technical detail of the design and results of these bugs (which are similar to our correctness testing) is left for the interested reader to explore.

5.5 Recovery Times

From the results in the figures above, we observe the recovery time varying linearly with the number of logged operations. The duration varied from 38-64 ms for 5 and 10 operations contained in each snapshot respectively. Translated into percentage overhead, this ranges from 14.6% to 26.0% with respect to application startup without the replay mechanism in place. Although these numbers seem prohibitively high, the absolute durations are still below the human perceived lag tolerance threshold of 100 ms [16].

6 Related work

There are a number of existing checkpoint and recovery solutions in industry. Among them is Assure which can recover from unknown software errors with the help of rescue points [23]. Rescue points are existing locations in the code for handling anticipated failures. When an unanticipated error occurs, execution is restored to the closest and most favourable rescue point. For instance, an out of bounds exception can be turned into ”return from function with error”. One can relate this to a transaction abort applied to functions. Any of the functions on the call stack can be rolled back to provide the best recovery and the execution then continues at the caller. Assure does not require code changes, is automated and can tolerate polymorphic inputs unlike Rx [19]. Rx is another checkpoint and recovery solution which tolerates both deterministic and nondeterministic faults by replaying operations in a modified execution environment.

Our solution does not prevent the application from crashing, contrary to other existing solutions discussed in this report such as Assure and Rx. Restarting applications is expensive for large production systems and in our case, interrupts the user's workflow. Another recovery solution is to partition the application into independent modules that communicate via a well defined protocol, similar to message passing systems [7]. When an error occurs, one or more of the modules affected by the fault is restarted instead of the entire system. Microreboot is one such system [4]. The obvious benefits of this is availability.

Pencil has well defined modules that handle different aspects of the application. Unfortunately, in its current form, it is not modularized in a manner that would take full advantage of microreboot. Although the Interface, Structure, Graphics and Plugin components can be made to restart independent of each other with minimal effort, the Interface components are tightly dependent on each other. For example, a single drawing operation pingpongs between methods in the ScribbleArea and the Editor. If the Editor fails, the entire Interface module would need to restart. This will also require an unwieldy amount of state to be saved as well for successful reintegration.

Application-level monitoring solutions like Chronus [25] and ConfAid [2] focus on identifying configuration token changes that cause the buggy behaviour, others such as X-ray [1] can undo the operations and recover a stable state prior to the crash or performance degradation. Below we will briefly discuss the contributions of each tool, and compare them to what we achieve with our solution.

Chronus captures disk-based modifications using snapshots (time disks) based on a user-based software state probe. With the aid of Virtual Machine Monitors, binary search and other tools (e.g. UNIX diff) it is able to show when the incorrect configuration or input value was introduced and what it is. In comparison to our solution, Chronus captures information at a coarser granularity by recording every write operation which is often prohibitive because of the associated overhead. For example, the removal of the Mozilla file tree produced 1432 MB of logs in addition to time disk size. Reconstruction of user data would take much longer. However, the overall method of capturing snapshots and operation logs is the same as ours. Searching through states would be applicable to future work on Pencil, if the recovery and bug finding process is to be automated. For instance, the application can automatically apply as many operations as possible without reintroducing the crash/bug.

ConfAid on the other hand instruments binaries and dy-

namically tracks causality, thus aiming to pinpoint when a configuration change (and even which token) causes a system misbehaviour such as a crash or error. Given an input binary, source of configuration and erroneous external output, it looks at control flow graphs, byte-level instructions and uses heuristics to deduce in a matter of minutes (on production server software) what change is responsible for the observed behaviour. For our use case, binary instrumentation is not necessary, and although ConfAid could identify erroneous program configuration changes, it will not be able to recover user work upon restart.

Similarly, X-ray attempts to solve performance degradation by not only determining the events causing performance anomalies but also the reason behind their occurrence. By recording checkpoints containing process system call data and timings/values of signals, it is able to deterministically replay events in the kernel by using these stored values in the program re-execution. This approach is similar to our solution, where we use canvas snapshots as checkpoints and logged operations for replaying events.

As described in the Purpose and Implementation Details section, our tool aims to save very specific data from the application to help remediate effects of crash bugs in addition to just configuration ones. Overall, although similar in methodology, logging (i.e. storage) and runtime overhead from using industry solutions described above are unnecessary, as they capture and process a lot more high level information than is useful for our purposes.

7 Future Work

In this section we provide a few possible directions in which the work presented in this report can be extended. One task we started but could not complete in time for this report is fully exposing the recovery process to the user and making it interactive. This is discussed in more detail in Section 7.1. We also implemented, tested and evaluated recovery for a subset of Pencil's drawing tools: Pencil, Pen and Eraser. The logging facilities can be extended to the other tools in a similar fashion. To enable support for the other tools, one simply needs to add operation logging code into the ScribbleArea method(s) that gets executed when the respective tool is used on the drawing surface.

Currently we are writing directly to disk at the end of every user event (mouse press, drag and release). Based on the evaluation results, we find this has considerable overhead. We can improve the performance by moving the logging out of the critical path of the I/O operation. That is, write the logging code to data structures in main memory and offload the task of writing from memory to disk to a separate thread. Similarly, we can also move the periodic snapshot to a separate thread that sleeps until it gets woken up when a snapshot needs to be taken.

During the recovery process the algorithm simply reperforms the draw strokes on the canvas. More elaborate processing such as including the event into the redo/undo data buffers, logging and submission of bug reports for detected faulty operations and other similar advanced techniques are left for future exploration.

Finally, a crash during the recovery phase was not considered in the evaluation presented in this report. However, we do not log operations or take snapshots during the restore operation. Additionally, the snapshot files gets removed in subsequent successful open/save project and program exit operations. This means if there was a crash during recovery, the user can restart the application with the snapshot data still intact and try to replay a smaller subset of operations, or restore a different snapshot.

7.1 User Interactive Recovery

After an unexpected termination of the program, on application restart, the Undo and Recovery feature should allow the user to select a snapshot ordered by timestamps from a list. The user will then be able to select which set(s) of operations, occurring after the selected snapshot, he/she would like to replay from those recorded in the log file. The operations presented will include only user events like a single paint stroke (i.e. user understandable actions). This will allow the user to redo the desired operations in real-time. Functionally, this is a stripped version of TimeWarp [6] where users can rewind, alter their histories and replay them at any time. However, TimeWarp performs rewind logically and we do it physically.

Besides giving users an interactive recovery experience, this feature was intended to mitigate the problem of crashing repeatedly during recovery due to replaying deterministic bugs. We did not want to implement a mechanism that would automate the detection of a bug and take appropriate actions during recovery, as it is not trivial and requires substantial programming effort. A generalized detection tool exists that relies on inferring programming rules from statically analyzing the code [8]. This allows it to explore all avenues that can cause a problem but suffers from path explosion and many false positives.

With respect to deterministic bugs, we offloaded the task to achieve successful recovery to the user by pruning out the operation that caused the bug by virtue of trial and error. Rx is an example of an automated checkpoint and recovery mechanism that dynamically alters the execution environment based on the failure to avoid deterministic or recurring errors [19]. Another example is Sweeper which uses taint analysis on external inputs to identify the one(s) that caused the failure [24]. Sweeper then creates an input filter to remove the culprit and any other similar future inputs, before rolling back to a checkpoint preceding the erroneous behaviour and doing a replay with the filter ap-

plied. This is another automated approach to repairing the environment that does not require human input, as was discussed before.

8 Conclusion

This report introduces an Application Level Undo & Recovery scheme to the Pencil open-source software. Periodic snapshots of the bitmap and vector animation layers along with logs of user-performed operations and internal application state is persisted to disk for recovery in case of a future crash. Upon normal program exit or successful user project opening or saving, logs are purged to reduce overhead. When an application crash occurs (due to a code bug or performance degradation stemming from configuration changes or other sources), the user would be able to select recently saved snapshots and operations to reapply in order to recover their work.

The current implementation uses existing Pencil snapshot functionality while introduces fine-grained draw operation logging to enable replay based recovery of user work. We have found, as expected, that runtime and storage costs vary linearly with the number of operations logged and the logging I/O costs can be partially hidden by caching log operations. Additionally, recovery and I/O times are not prohibitively high so as to discourage the user from using this safety guard. We find 20-30 ms processing per mouse movement in a discrete operation, 7-10 kB of storage space per discrete draw operation per second and 4-6 ms per discrete event for recovery on startup. This is in addition to existing snapshot overheads of approximately 8-10 ms processing and 2-3 kB storage for each discrete draw event. As a result, our solution is a worthwhile addition to Pencil in helping preserve the best drawing experience for the end user in the face of unexpected crashes from programmatic, environmental and other unforeseen bugs.

References

[1] Mona Attariyan, Michael Chow, and Jason Flinn. Automatic root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX conference on Operating systems design and implementation*, OSDI'12, pages 1–12, Berkeley, CA, USA, 2012. USENIX Association.

[2] Mona Attariyan and Jason Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–11, Berkeley, CA, USA, 2010. USENIX Association.

[3] Aaron B. Brown and David A. Patterson. Undo for operators: building an undoable e-mail store. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '03, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[4] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[5] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, 2000.

[6] W. Keith Edwards and Elizabeth D. Mynatt. Timewarp: techniques for autonomous collaboration. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*, CHI '97, pages 218–225, New York, NY, USA, 1997. ACM.

[7] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[8] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[9] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems 1986*, pages 3–12, 1986.

[10] VMware Inc. Vmware virtualization software for desktops, servers & virtual machines for public and private cloud solutions:, 2012. [Online; accessed 4-December-2012].

[11] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 20–20, Berkeley, CA, USA, 2000. USENIX Association.

[12] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[13] James Newsome, David Brumley, and Dawn Xiaodong Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *NDSS*. The Internet Society, 2006.

[14] US Department of Commerce NIST. National vulnerability database (nvd) cve statistics:, 2006. [Online; accessed 5-December-2012].

[15] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 1–11, New York, NY, USA, 2007. ACM.

[16] Lothar Pantel and Lars C. Wolf. On the impact of delay on real-time multiplayer games. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, NOSSDAV '02, pages 23–29, New York, NY, USA, 2002. ACM.

[17] Patrick Corrieri Pascal Naidon. Pencil - a tranditional 2d animation software, 2009. [Online; accessed 4-December-2012].

[18] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[19] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 235–248, New York, NY, USA, 2005. ACM.

[20] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 82–90, Washington, DC, USA, 2004. IEEE Computer Society.

[21] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *NDSS*. The Internet Society, 2004.

[22] Stelios Sidiroglou, Giannis Giovanidis, and Angelos D. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *Proceedings of the 8th international conference on Information Security*, ISC'05, pages 1–15, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: automatic software self-healing using rescue points. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 37–48, New York, NY, USA, 2009. ACM.

[24] Joseph Tucek, James Newsome, Shan Lu, Chengdu Huang, Spiros Xanthos, David Brumley, Yuanyuan Zhou, and Dawn Song. Sweeper: a lightweight end-to-end system for defending against fast worms. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 115–128, New York, NY, USA, 2007. ACM.

[25] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[26] A.P. Wood. Software reliability from the customer view. *Computer*, 36(8):37–42, 2003.

[27] Chris Wysopal. Veracode: State of software security. 4, 2011.