# Self-Recovery in Server Programs

Vijay Nagarajan, Dennis Jeffrey and Rajiv Gupta
University of California, CSE Department, Riverside, CA 92521
{vijay,jeffreyd,gupta}@cs.ucr.edu

## Abstract

It is important that long running server programs retain *availability* amidst software failures. However, server programs do fail and one of the important causes of failures in server programs is due to *memory errors*. Software bugs in the server code like buffer overflows, integer overflows, etc. are exposed by certain user requests, leading to memory corruption, which can often result in crashes. One safe way of recovering from these crashes is to periodically checkpoint program state and *rollback* to the most recent checkpoint on a crash. However, checkpointing program state periodically can be quite expensive. Furthermore, since recovery can involve the rolling back of considerable state information in addition to replay of several benign user requests, the throughput and response time of the server can be reduced significantly during rollback recovery.

In this paper, we first conducted a detailed study to see how memory corruption propagates in server programs. Our study shows that memory locations that are corrupted during the processing of an user request, generally do not propagate *across* user requests. On the contrary, the memory locations that are corrupted are generally *cleansed* automatically, as memory (stack or the heap) gets deallocated or when memory gets overwritten with uncorrupted values. This self cleansing property in server programs led us to believe that recovering from crashes does not necessarily require the expensive roll back of state for recovery. Motivated by this observation, we propose SRS, a technique for self recovery in server programs which takes advantage of *self-cleansing* to recover from crashes. Those memory locations that are not fully cleansed are restored in a demand driven fashion, which makes SRS very efficient. Thus in SRS, when a crash occurs instead of rolling back to a safe state, the crash is suppressed and the program is made to execute *forwards* past the crash; we employ a mechanism called *crash suppression*, to prevent further crashes from recurring as the execution proceeds forwards. Experiments conducted on real world server programs with real bugs, show that in each of the cases the server program could efficiently recover from the crash and the faulty user request was isolated from future benign user requests.

***Categories and Subject Descriptors*** D.4.5 [*Operating Systems*]: Reliability – Checkpoint/Restart, Fault-tolerance

***General Terms*** Reliability, Performance, Experimentation

***Keywords*** self cleansing, self recovery, memory propagation



*ISMM'09,* June 19–20, 2009, Dublin, Ireland.


## 1. Introduction

Long running server programs seek to maximize their uptime and thereby ensure that they are *available* to users. However, server programs do fail and one of the important causes of failures in server programs is due to *memory errors*. According to the *National Vulnerability Database* [nvd], memory errors like buffer overflows, format string errors, integer overflows etc. constitute a significant percentage (30% as of 2008) of software failures. Memory bugs in the server code, when exposed by certain user request, can lead to memory corruption which can eventually lead to crashes or even software attacks (if user input is malicious).

There has been significant research on the recovery from software failures. One safe way of recovering from such software failures and ensure availability is to periodically checkpoint program state and rollback to the most recent checkpoint, when failure is detected [Gray 1986, Plank et al. 1998, Qin et al. 2005, Randell et al. 1978, Tallam et al. 2008]. Having rollbacked to a safe state, all user requests starting from the safe state point until the crash point are replayed; during replay, the particular bad user request that triggered the crash is identified and dropped [Qin et al. 2005, Tallam et al. 2008] so that the same failure is not repeated. However the checkpointing/rollback scheme has its limitations. First, checkpointing program state periodically can be quite expensive. Second, since recovery can involve rolling back of considerable state information, the throughput and response time of the server can be reduced significantly during rollback recovery. Third, since checkpointing is done only at specific program points, recovery can involve the replay of several good user requests. Finally, checkpointing/rollback system is complex and poses implementation challenges for multithreaded programs [Qin et al. 2005].

In this paper, we first conducted a detailed study of memory corruption in server programs, to see how memory corruption propagates as the server program executes, with a view to understand whether the expensive checkpointing and rollback operations are really needed. If the memory corruption does really propagate a lot through memory, then that would vindicate the expensive state rollback in the checkpointing/rollback approach; on the contrary, if memory corruption does not propagate all that much, such expensive recovery mechanisms might not be really needed. In our study of real world server programs, we assumed that the values written to the memory by different store instances (we comprehensively tested all store instances) to be "corrupt" and studied its propagation. From this study, we found that memory locations that are corrupted during the processing of an user request, generally *do not propagate across user requests*. On the contrary, the memory locations that are corrupted are often *cleansed* automatically, as memory (stack or the heap) gets deallocated or when memory gets overwritten with uncorrupted values. Thus this self cleansing property in server programs, led us to believe that recovering from crashes does not necessarily require the expensive roll back of state for recovery.

Motivated by the above study, we propose *SRS* a safe technique for enabling Self Recovery in Server programs. In SRS, we take advantage of the self-cleansing property in server programs to *isolate* the faulty user request from other succeeding benign user requests, without checkpointing or rollback. In other words, in SRS, when a crash occurs, we do not rollback state to a previously saved safe state; on the contrary, we *suppress the crash and execute forward*. When executing forward, SRS guarantees that the user requests succeeding the faulty request do not read any values written during the processing of the faulty request, thereby achieving the effect of dropping the bad request. Despite self cleansing, there can be a small number of memory locations that remain corrupt; if there is a need to access such a corrupted location, we use a *demand driven* approach to restore the corrupted value when a read for it is encountered. Thus in SRS, instead of the expensive rollback operation to restore the memory contents to a safe state, we use the efficient *demand driven restoration*. Furthermore, since execution is made to proceed forward past a crash, the need to replay benign user requests is eliminated altogether.

However, executing forwards past a crash is not without its own challenges. Even though the first crash can be suppressed, similar crashes can recur (and likely will recur), when values that are dependent on the corrupted memory values are used later. To prevent such crashes from recurring we use a mechanism called *crash suppression* in which those instructions that use values that are corrupted are not made to execute, and are *suppressed*. Owing to the self-cleansing property, fewer memory locations remain corrupted because of which fewer instructions need to be suppressed as execution moves forward.

We evaluated SRS with real world memory bugs in 4 widely used server programs and found that SRS could successfully recover from the failures caused by faulty user requests. We also found that SRS is efficient, causing negligible drop in the response time of the program during normal run and after recovery.

Thus the main contributions of this paper are as follows:

- *Self Recovery.* We present a self recovery technique called SRS that hinges on the *self-cleansing* property inherent in server program.
- *Efficient Recovery through demand driven restoration.* In SRS, we do not have the expensive checkpointing or rollback operations. On the contrary, the *self cleansing* property allows us to execute *forwards* past a crash, and restore the residual corrupted values using *demand driven restoration*.
- *Crash Suppression.* To enable execution forward past a crash, we use a mechanism called, crash suppression where instructions which use corrupted memory values as its operands are suppressed and not executed.

This paper is organized as follows. In section 2 we describe the detailed study we conducted to study how memory corruption propagates in server programs. Motivated by the observations from the study we propose and describe our technique for self-recovery, SRS, in section 3. In section 4 we evaluate SRS with real world bugs in server programs and also evaluate the overhead imposed by SRS. In section 5, we discuss related work and finally, we conclude in section 6.

## 2. Study of Memory Corruption Propagation

The process of exposing a memory bug in a program via a user request that causes a crash consists of three events in program execution. In the first step, the bug in the source code is traversed by user input. In the second step, the traversal of the bug leads to the first point of memory corruption; this is the point when a memory location is mishandled in some way. The corrupted memory location then *propagates* across memory where it spreads and corrupts other memory locations. Finally, a crash occurs when there is an access to a spurious memory location. The goal of a checkpointing/rollback system is to rollback to a prior memory state, a state that is hopefully devoid of memory corruption. We conducted a detailed study of memory corruption in server programs to see how memory corruption propagates as the server program executes, with a view to understand whether the expensive checkpointing and rollback operations are really needed. Does the corrupted memory location go on to corrupt other memory locations; if so on an average how much does the corruption spread? Is it possible that the set of corrupted memory locations can shrink? These were some of the questions we wanted to answer with our memory propagation study. If the memory corruption does really propagate a lot through memory, then that would vindicate the expensive state rollback in the checkpointing/rollback approach; on the contrary, if memory corruption does not propagate all that much, such expensive recovery mechanisms might not be really needed. In this section, we first discuss the server programs we used in our study. Then we describe the study of memory corruption propagation along with an analysis of salient observations.

**Table 1.** Server Programs Characteristics.

| Program | Description | LOC |
|---|---|---|
| mysqld | Database server | 588K |
| cvs | Version Control server | 93K |
| squid | Web Proxy cache server | 283K |
| Apache | Web server | 114K |

We used 4 widely used real world server programs in our study, *mysqld*, *cvs*, *squid*, and *apache* as listed in Table 1. For each of these programs, the client sends separate user requests; for *mysqld* we send 3 separate requests that build and manipulate separate tables; for cvs we send 3 separate requests to checkout source code; for *apache* and *squid* we send 3 requests to download files of various sizes.

### 2.1 Memory Propagation Study

**Methodology:** The purpose of this study is to understand how memory corruption propagates through memory as the program executes. One possibility is to study the propagation of the memory corruption in real bugs. However, the memory propagation clearly depends on what memory location(s) are corrupted. For example, a global variable that is marked corrupt is expected to lead to several other corrupted values, while a local temporary may only lead to fewer corrupt values. Hence, to comprehensively study the propagation of memory corruption, we consider several executions of the server program; in each execution, we consider different memory locations to be corrupt and study their propagation. More specifically, in each execution we assume that a *specific dynamic store instruction writes a corrupt value to memory and study the propagation of this corruption in the rest of the execution*. This way, we exhaustively cover all possible memory locations that can potentially be corrupted.

The steps of the memory propagation study, as illustrated in Fig. 1, consists of two phases. In the first phase, called *Trace Stores*, we collect a trace of all store instructions and the corresponding instruction count when each store instruction is executed (see step 4 in 1st phase), so that each dynamic store can be uniquely identified. In the second phase, called *Memory Corruption Propagation*, we repeatedly assume each unique dynamic store to be corrupted, i.e. we *assume* that it writes a corrupt value to the memory, and study the memory corruption propagation. This is achieved by associating a *corruption* bit with every memory word and register

**Table 2.** Memory Corruption Propagation.

| Action | Target | Src1 | Src2 |
|---|---|---|---|
| Fault | Corr. | | |
| Instr: Target = src1 op src2 | Corr. | Uncorr. | Corr. |
| Instr: Target = src1 op src2 | Corr. | Corr. | Uncorr. |
| Instr: Target = src1 op src2 | Corr. | Corr. | Corr. |
| Instr: Target = src1 op src2 | Uncorr. | Uncorr. | Uncorr. |
| Deallocation | Uncorr. | | |

```
Phase 1: Trace Stores

Let cnt: global instruction count

1.  switch (instruction)
2.    cnt ++                              // Update Current instruction count
3.    case store:
4.      fprintf (trace-file, cnt)         // Write current instruction count to trace file

Phase 2: Memory Corruption Propagation

For each memory word addr, register reg,
src.corruption : Whether src (address/reg) is corrupted

1.  while ( !eof (trace-file) )
2.    fscanf (trace-file, propagate_cnt)    // Read count from trace file and start new execution
3.    switch (instruction)
4.      cnt++                               // Update current instruction count
5.      case target = src1 op src2          // Target, src1, src2 can be memory / register
6.        if (cnt == propagate_cnt)
7.          propagation = true              // Start propagation mode
8.          target.corruption = true        // Initial Corruption
9.        if (propagation)
10.           if (src1.corruption or src2.corruption)
11.             target.corruption = true    // Propagation semantics
12.       target = src1 op src2             // Regular semantics
```

**Figure 1.** Algorithm for Memory Propagation Study.

and propagating the *corruption* bits as the program executes. using the propagation rules detailed in Table. 2 (see Fig. 1 2nd phase, steps 9,10). When an instruction experiences an exception (for example an out-of-bounds load), the target of that faulting instruction is marked corrupt. When a corrupted location is used by an instruction, then the value computed (or defined) by that instruction in turn is marked corrupt. However, when an instruction whose uses are uncorrupted, redefines a value, then the redefined value is considered to be uncorrupt, since it is computed with valid operands. Finally, when memory (stack or heap) gets deallocated it is marked uncorrupt. Thus for a *load* instruction that moves a memory value into a register, the corruption bits corresponding to the memory word are also moved into a register. Similarly, the corruption bits are cleared when memory gets deallocated.

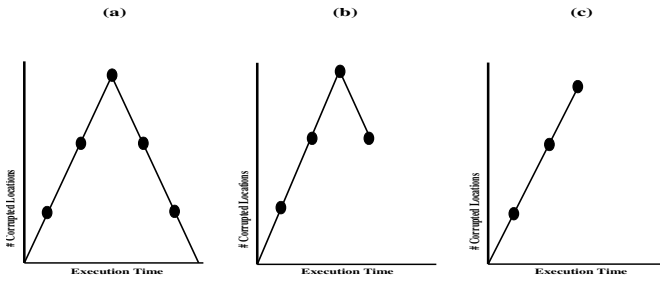


**Figure 2.** Variation of Corrupted Memory Locations with Time.

**Observations and Analysis:** When a memory value is corrupted, it can go on to corrupt other memory locations; at the same time, the memory locations that are marked corrupt can revert back to being uncorrupt, as memory gets deallocated. Fig. 2 shows different ways in which the number of corrupted memory locations varied, after the initial memory corruption. One common pattern we observed was that, a memory location once corrupted, *marks zero or more memory locations* corrupt, *each of which* revert back to uncorrupted state, as that respective memory locations are deallocated (see Fig. 2(a)). On the other hand, Fig. 2(b) shows the situation in which not all of the memory locations that are corrupted, revert back to uncorrupt state. Similarly, Fig. 2(c) shows the situation in which none of the memory locations that are corrupted, revert back to uncorrupted state. This can happen when the corrupted variables are those variables that are used *across user requests*, in which case they are not deallocated.

**Table 3.** Memory Propagation Study.

| Program | Max Corrupted Location | | | | Final Corrupted Locations | | | |
|---|---|---|---|---|---|---|---|---|
| | Min | Max | Mean | Med | Min | Max | Mean | Med |
| mysqld | 1 | 14241 | 377 | 1 | 0 | 4995 | 120 | 0 |
| cvs | 1 | 1672 | 56 | 1 | 0 | 577 | 18 | 0 |
| squid | 1 | 475771 | 6997 | 2 | 0 | 8680 | 423 | 0 |
| apache | 1 | 32672 | 3228 | 5 | 0 | 6631 | 607 | 0 |

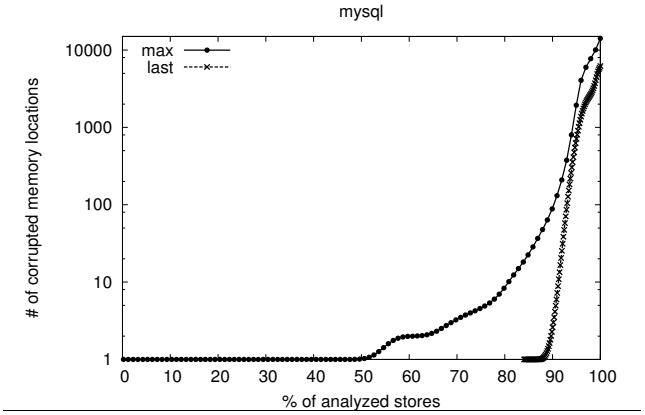


**Figure 3.** Variation in Max Corrupted and Final Corrupted across different execution instances for *mysqld*.

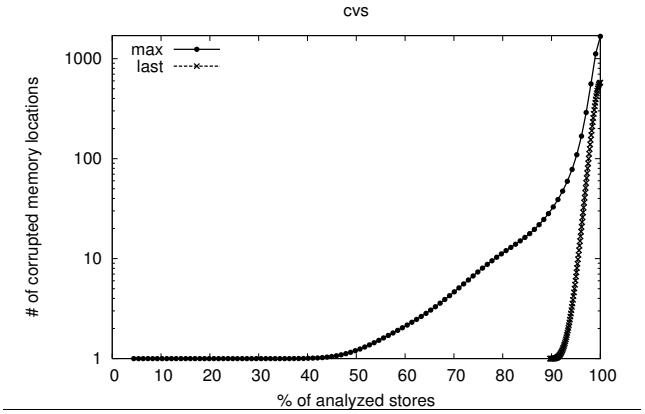


**Figure 4.** Variation in Max Corrupted and Final Corrupted across different execution instances for *cvs*.

To get a quantitative perspective of the propagation of memory corruption, we measured *the maximum number of corrupted memory locations* and *the final number of corrupted locations at*

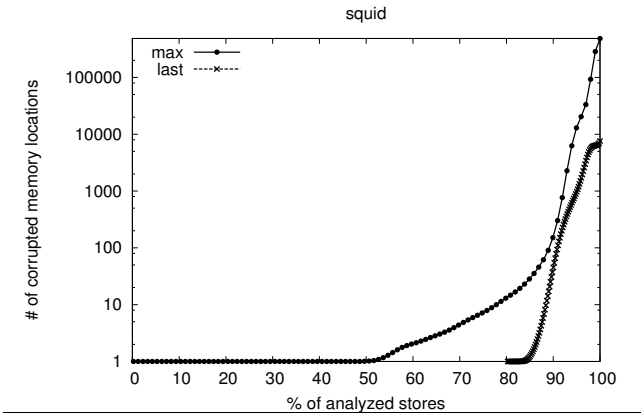


**Figure 5.** Variation in Max Corrupted and Final Corrupted across different execution instances for *squid*.

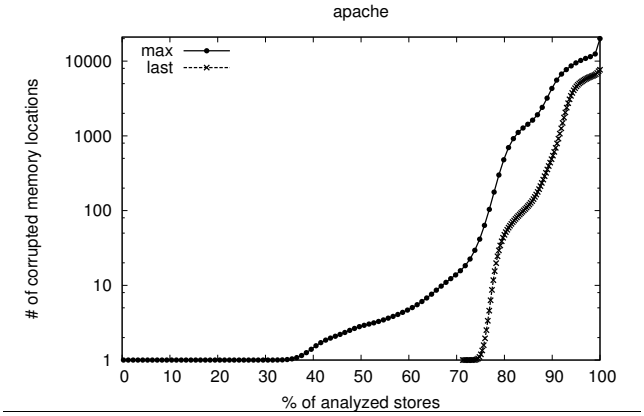


**Figure 6.** Variation in Max Corrupted and Final Corrupted across different execution instances for *apache*.

*the end of the current user request*. For each of the above metrics we measured the *min, max, mean and median* across different executions, where different memory locations are marked corrupt as shown in Table. 3. To study how the above values are distributed across different executions, we also plotted the values for each of the benchmarks as shown in Figs. 3 through 6. Each graph shows the variation in the max corrupted values and final number of corrupted values over different execution instances, in each of which an unique store is assumed to write a corrupt value. From Table. 3 and Figs. 3 through 6 we make the following observations:

- The minimum value of maximum number of corrupted locations (minmax) across all benchmarks is 1; this is the one that is initially marked corrupt and does not corrupt any more new locations. This can happen for several reasons. One possible reason is because the corrupted value can be used as a loop counter, in which case the corruption does not propagate across other memory locations. Another possible reason is because the stored value is sometimes not read at all, in which case no propagation occurs.
- We also observe that the minimum value of final number of corrupted locations (minfinal) is 0 . This corresponds to the case in which each of the memory locations that were marked corrupt were reverted back to uncorrupted state. This can happen due to two reasons. First, those memory locations that have been marked corrupt could have been deallocated; second, the values that have been marked corrupt could have been overwritten with uncorrupt values.
- The maximum number of maximum number of corrupted locations (maxmax) can be large. This can happen if some of the important variables that has several uses is marked corrupt. However, as we can see from Figs. 3 through 6 often the maximum number of corrupted locations is quite low. For 80% of the execution instances, less than 10 memory locations get corrupted.
- We also observe that the maximum number of final number of corrupted locations (maxfinal) is relatively lower than maxmax. This is because of what we call the *self cleansing* effect in server programs; this causes a large number of memory locations that are marked corrupt are reverted back to uncorrupt state. This fact is confirmed from Figs. 3 through 6 where the final number of corrupted locations is significantly lower. In fact, between 70% and 90% of the execution instances, the final number of corrupted memory locations is 0, meaning whatever memory locations that were marked corrupt were fully cleansed.
- This fact is further reinforced when we observe the median of the maximum number of corrupted locations (medmax) and final number of corrupted locations (medfinal). While medmax is *less than 5* across all benchmarks, medfinal is 0. This means that more often than not, a corrupted memory location goes on to mark only few other additional memory locations as corrupt, each of which are reverted back to uncorrupted state.

Thus the most important insight that we inferred from the memory corruption propagation study is that a memory location that is marked corrupt goes on to corrupt only a few other memory locations, most of which are uncorrupted by the end of processing of the user request. This is what we call the *self cleansing* property inherent in server programs.

### 2.2 What causes self cleansing?

Next, we wanted to study and find out the reasons for the above observations. In particular, we wanted to figure out *why* self cleansing takes place in server programs. Why does memory corruption spread and then diminish rapidly as the server begin to handle the next request. One possible reason could be that user requests are already isolated, in that, there is very little data that is shared between user requests, which can cause memory locations corrupted during the processing of one user request to become invisible for other user requests. So, we conducted this study if this is indeed true.

```
Memory Isolation Study

Let curr-id : Current user request id
For each memory word addr:
addr.request-id: User request that wrote to it last
shared : set of shared memory locations

1.   switch (instruction)
2.      case store:
           addr.request-id = curr-id
3.      case load:
           if (addr.request-id < curr-id)
              shared = shared U addr
```

**Figure 7.** Algorithm for Isolation Study.

**Methodology:** The purpose of this study is to determine the degree of isolation among user requests already inherent in server programs. In this study, we connected the server with several user

**Table 4.** Isolation Study.

| Program | # Shared | # Non-Shared | % of Shared |
|---|---|---|---|
| mysqld | 758 | 6493 | 10.4 |
| cvs | 271 | 1457 | 15.6 |
| squid | 2753 | 41650 | 6.2 |
| Apache | 1471 | 2646 | 35.7 |

requests, and determine the number of memory locations that are *shared* across user requests. A memory location is said to be shared if it was written into by an earlier user request, and read by a later request; in other words if a memory location is used to exercise an inter user request RAW dependence, it is considered shared. A small percentage of shared memory locations, would mean that values written during the processing of one request, is not read during the processing of other user requests and thus can explain why *self cleansing* takes place. To determine the number of shared memory locations, we instrument stores and loads as shown in Fig. 7. Each memory location is tagged with a shadow memory location; each store is instrumented to store the current user request id in the shadow memory location associated with the original address. Each load is instrumented, to check if the value loaded comes from a previous user request, by checking the user request id; if that is the case, that particular memory address is added to the set of shared addresses.

**Observations and Analysis:** We measured the percentage of memory locations that are shared across several user requests in each of the server programs. As we can see from Table. 4 only a small percentage of memory locations (ranging from 6% to 35%) were shared across user requests. We also measured the effect of the complexity of user requests on the number of shared memory locations. We present the results for two programs *mysqld* and *squid*. For *mysqld*, we measured the number of shared memory locations, as the size of the tables were varied. For *squid*, we varied the sizes of the webpages that squid fetches and measured the effect on the number of shared memory locations. As we can see from Figs. 8 and 9: as the complexity of user requests increases, the number of non-shared memory locations increases rapidly, while the number of shared memory locations almost remains the same. This evidence points to the fact that server programs have a fixed *global state* that is shared across user requests.

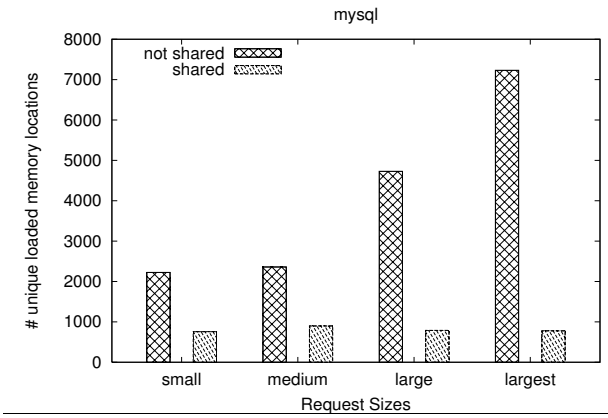


**Figure 8.** Variation in Shared/Unshared with complexity of user requests for *squid*.

Thus the most important insight we derived from this study is that most of the values that are written during the processing of a user request are used locally; only a small (fixed) amount of global state is shared across user requests. The small shared state is the reason why memory corruption does not propagate across user requests, and hence explains self cleansing.

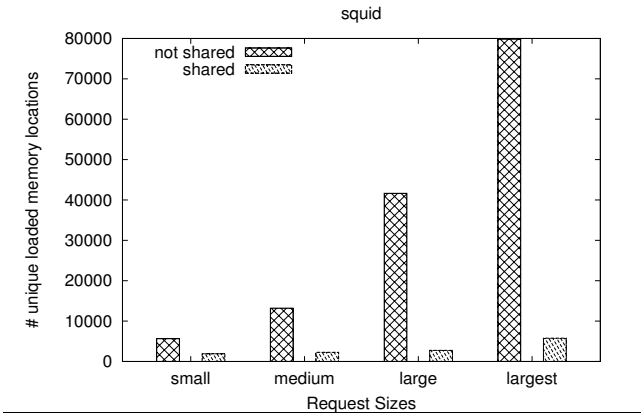


**Figure 9.** Variation in Shared/Unshared with complexity of user requests for *mysqld*.

## 3. Design and Implementation of SRS

In this section, we describe in detail the design and implementation of our technique SRS, for recovering from server failures related to memory corruption. First, we discuss the general idea of SRS at a high level. Then we discuss in detail the steps involved in implementing SRS at a conceptual level.

When a server program experiences a failure while processing a user request, ideally the server should not crash; on the contrary, the server program should continue to process future user requests. At the same time, the memory state that has been corrupted by the fault inducing memory request *should not* be visible to future user requests. In other words, the faulty user request should be *isolated* from the processing of other benign user requests. One way to perform this is to checkpoint memory and architectural state before processing every user request; upon the detection of a failure we can rollback to the prior benign state. The objective of SRS is to ensure semantics similar to rollback based recovery scheme, without performing checkpointing or rollback. For this SRS takes advantage of *self cleansing* inherent in server programs. Recall that from the study we learned that different user requests of sever programs share very little shared state, owing to which memory corruption that happens during the processing of a user request is largely invisible to future requests. Thus the main steps in SRS are two fold:

- Execute past a failure, when a failure is detected, instead of rolling back, so as to trigger *self cleansing*. We enable execution past a failure by executing instructions under *crash suppression* mode – under suppression mode, instructions, any of whose source operands are corrupt, are *not executed* and the target is *marked corrupt*.
- Despite self cleansing there can be small number of memory locations that remain corrupted – if there is a need to access such corrupted locations later (by a later request), we use a *demand driven* approach to restore the corrupted values only when it is needed to be read. In SRS, the set of memory locations that are potentially shared across user requests are identified via profiling; these specific memory locations are specially monitored, so that in case of a failure *these specific locations are restored in a demand driven fashion*.

The rest of this section is organized in follows. We first describe in detail the crash suppression semantics and then discuss how we implement it. Then we discuss how we realize isolation with demand driven restoration of shared memory locations. Then we briefly discuss how SRS maintains thread safety for multithreaded code. Finally, we integrate each of the above steps and present as a whole, the SRS technique.

```
For each memory word addr, register reg
   src.corruption : Whether src (address/reg) is corrupted

1.   switch (instruction)
2.      case faulting instruction:
3.         suppression = true                  // Start suppression mode
4.         target.corruption = true            // Initial Corruption

5.      case target = src1 op src2
6.         if (suppression)

7.            if (src1.corrupt  or src2.corrupt)
8.               target.corrupt = true         // Suppression semantics
9.            else
10.              target = src1 op src2         // Regular semantics
```

**Figure 10.** Suppression Semantics.

### 3.1 Crash Suppression

Once a failure is detected while processing a user request, in SRS, we continue execution forwards so that we can *take advantage of the self-cleansing property*[1]. However, executing forwards past a crash is not without its own challenges. Even though the first crash can be suppressed, similar crashes can recur (and likely will recur), when values that are dependent on the corrupted memory values are used later. We enable execution past a failure using *crash suppression semantics*, in which an instruction is *suppressed* without executing, if any of its source operands are corrupted. The basic steps for realizing *suppression* semantics are outlined in Fig. 10. We implement suppression by associating a *corruption* bit with every memory word and register. Upon detecting a failure, the *corruption* bit for the target of the instruction that causes the failure is set. Suppression semantics entail that any instruction, any of whose source operands are marked corrupt, is suppressed and is not executed. However, the corruption bit is propagated, which means the target operand's corruption bit is set. If the corruption bit for a branch predicate is set, the control is made to skip the whole branch structure; For instance if the predicate for an *if-then-else* structure is marked corrupt, then both the *then* and the *else* parts of the structure are skipped. Dealing with indirect jumps (and returns) whose target address is marked corrupt is more tricky. In our current implementation, we use profiling to figure out the most frequent branch target for such indirect jumps and jump to that target. In case profile information is not available, we directly jump to the next user request.

### 3.2 Ensuring Isolation

We need to ensure that an user request, that encountered a failure, should be isolated from future benign user requests. This entails that values written to memory during the processing of the faulty request should not be visible to future requests. Those memory locations that are shared across user requests are those which can possibly be visible to future requests. Fortunately, our study shows that the number of shared memory locations are relatively small. In SRS, we identify memory locations that are likely shared using profiling. During normal run, these shared memory locations are monitored and multiple versions of these shared memory locations are maintained. Thus, when an user request experiences a failure and there is a need to access one these shared memory locations, we can restore the corrupted value to its original uncorrupted state, on demand. However, it is worth noting that profiling only gives us an *underestimate* of the set of memory locations that are shared across user requests. Thus, we could potentially encounter a situation when we need to access a corrupted shared memory location, which has not been identified and monitored. Our approach to deal with this situation is to provide capability to *detect* such a situation. When such a situation arises, we restart the server; by doing this we ensure that recovery is *fail safe*. Having explained our approach for ensuring isolation at a high level, now let us consider in detail the individual steps.

#### 3.2.1 Monitoring Shared Locations

We find the memory locations that are potentially shared across user requests using profiling. In the profiling run we connect to the client with several user requests, and then identify the memory locations which are written to by an earlier request and read by a later request. In other words, we identify the set of memory locations which are used to enforce *true dependences* across user requests. These are the memory locations in which global state is maintained. For instance, a global variable which essentially stores the *number of user requests handled* will have this property. Let us call these set of memory locations as the *TrackSet*. It is worth noting that this is only an underestimate, since we use profiling to identify this set. We do not need to worry about anti and output dependences; this is because these locations are overwritten with a new (uncorrupt) value in the later request.

We now discuss how the TrackSet is monitored and how different versions are maintained for the memory locations in the TrackSet. Whenever we start processing a new user request, we allocate memory, the *TrackLog* to hold the previous values of the memory locations within the TrackSet. The TrackLog is a buffer, each entry having two values: the address and the (previous) value. We instrument all stores operating on the TrackSet to maintain their prior values in the TrackLog. If the current user request does not experience a fault, then the TrackLog can simply be discarded. However, if the user request experiences a fault, the TrackLog is used later to restore the corrupt memory locations to their prior values, on demand.

#### 3.2.2 Demand Driven Restoration

The main idea of demand driven restoration, is to restore the corrupted value using the TrackLog, when it is is about to be read in a future request. To be able to do this we first need to identify that a value is corrupt, when it is about to be accessed. A value is considered corrupt, if it was written into during the processing of a fault inducing user request. To identify such memory values, we associate (or *shadow*) every memory location[2] with a *request-id*. This is essentially a unique number associated with every user request. All stores are then instrumented to additionally write the *current request-id* to the shadow location associated with the memory address.

When a user request experiences a failure, we remember this by adding the current request-id to the list of *failed requests*. Once a failure is encountered, all loads in the program are instrumented

[1] For this work, failure refers to an OS *out-of-bounds exception*, although others sensors [Qin et al. 2005] can be used to detect a failure

[2] It is sufficient if we shadow the heap and global space, since stack memory locations are not used to enforce true dependences across user requests

to perform an additional check. By comparing the request-id associated with the loaded memory location with the list of failed user request, we are essentially checking if the value loaded comes from a value that has been stored during the processing a fail request. If so, we then consult the TrackLog using the effective address as an index into the TrackLog. An entry in the TrackLog means that this memory location has been identified during profiling and the value has been backed up in TrackLog. Accordingly, we then restore the value from the TrackLog into the actual memory location, and use this restored value. In other words, we restore corrupted values on demand, when they are accessed.

However, it is important to note that the TrackLog is only an underestimate of the actual set of values that is shared across user requests. Hence it is possible that the TrackLog does not have an entry if the memory location had not been identified as a part of TrackSet during profiling. If this is the case, we then restart the server to ensure fail safety.

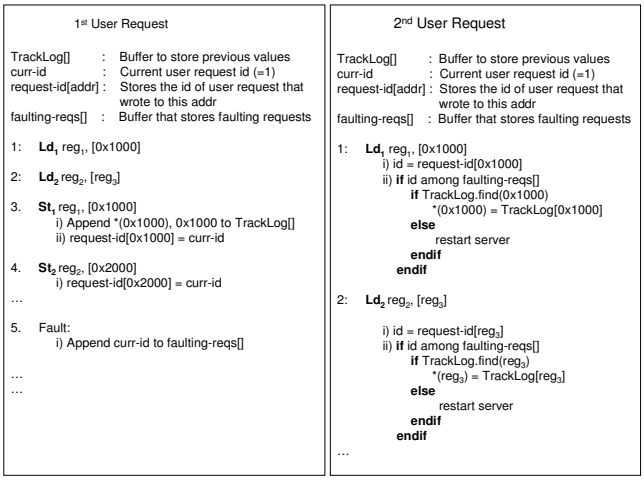


**Figure 11.** Ensuring Isolation.

### 3.2.3 An Example

In this section, we illustrate with a simple example which summarizes the steps involved in ensuring isolation. Let us consider two user requests the first of which experiences a fault while being processed. The first step is to identify the *TrackSet* using profiling. Let us assume that during profiling run, there is only one memory location ($0x1000$) that is shared between two user requests, and it is the one that is written into by $St_1$ and read by $Ld_1$ across user requests as shown in Fig. 11. Consequently, $St_1$ is instrumented with code that stores the prior value residing in the memory location to the TrackLog associated with the user request (step 3). Since $St_2$ does not write to the TrackSet, it is not instrumented in the above fashion. However, it is instrumented to write the current request-id in the shadow memory associated the memory location (step 4). Let us assume that the program then experiences a fault while executing a subsequent instruction, while processing the same user request. Upon a fault (step 5), the current request-id is added into the list of failed user requests.

Now let us consider the processing of the next user request, and in particular the execution of the two loads $Ld_1$ and $Ld_2$. $Ld_1$ which obtains its value from $St_1$, now gets it value from a store that was executed during the processing of a fault inducing request (request 1). In other words, the value to be loaded is corrupt and the condition (step ii) evaluates to true. Since the value is corrupt, the TrackLog is searched for the value. As the value was already appended in step 3 of the 1st request, it is found there. Consequently, the value is restored from the TrackLog and the correct (uncorrupt) value is loaded. Now, let us consider the execution of $Ld_2$. Let us assume that during this run $Ld_2$ actually gets its value from $St_2$ since the value of $reg_3$ happens to be $0x2000$ during this run. This is an instance of a dependency across user requests that was not captured during profiling. Consequently, the value is not found in the TrackLog and the server is restarted.

### 3.2.4 Handling Multithreaded Code

Since the SRS technique is associated with meta data for every memory location (for eg. request id) and includes software instrumentation associated with loads and stores, races present in the source can lead to meta data inconsistency [Chung et al. 2008]. In SRS, we deal with this problem by serializing the threads and making sure that thread switches do not occur between data and meta data updates as in [Nethercote and Seward 2007b]. However it is worth noting that thread serialization is inefficient as it forces the code to run on a uniprocessor. Solutions proposed in [Chung et al. 2008] and [Nagarajan and Gupta 2009] can be used to deal with meta data inconsistency problem without sacrificing the efficiency.

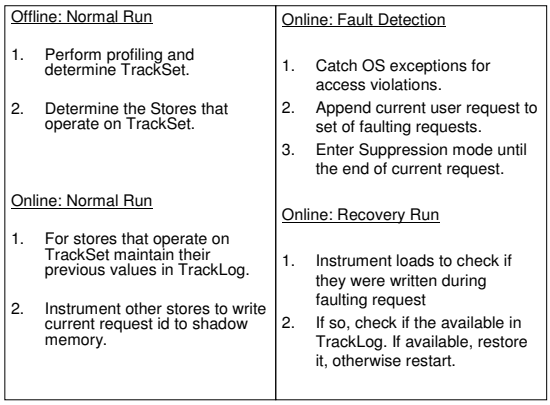


**Figure 12.** Summary: SRS.

## 3.3 SRS Summary

We now summarize each of the steps of SRS in the concise algorithm shown in Fig. 12. The steps of SRS are roughly divided into four phases. In the first profiling phase, which is performed offline, the *TrackSet* is determined and all the stores that write to the TrackSet are determined. In the normal run, the TrackLog is maintained for all stores that write to the TrackSet. For all other stores, the current request id is written out to the shadow memory to assist on-demand restoration. It is worth noting that the online overhead imposed by SRS is the overhead of executing the additional instrumentation. The next phase consists of the fault detection. In this phase, we use the OS *out-of-bounds exception* to denote a fault. However it is worth noting that we can use other sensors including security attack detection tools in this step. Once the fault is detected, the current user request is added to the set of faulting requests, and execution enters suppression mode. In the final recovery phase, loads are instrumented to check if the request id corresponding to the loaded value comes from a faulty user request. If this is the case, then the TrackLog is searched and the correct value is restored on demand. If the entry is not found in the TrackLog, the server is restarted.

**Table 5.** Bugs in Server Programs.

| Program | Bug |
|---|---|
| mysqld | Uninitialized Read [sql] |
| cvs | Double free [Lu et al. 2005] |
| squid | Buffer overflow [Lu et al. 2005] |
| Apache | Stack Overflow [Lu et al. 2005] |

## 4. Experimental Evaluation

We performed the experimental evaluation of SRS with several goals in mind. First and foremost, we wanted to see if SRS can recover from real faults in server programs. At the same time, we wanted to see what are the overheads imposed by SRS in the normal run and during recovery. But before we present our results, we briefly describe how we implemented the prototype for SRS.

### 4.1 Implementation

We implemented a functionally working version of SRS in *Valgrind* [Nethercote and Seward 2007b], which we also used for conducting the study. We used Valgrind's shadow memory support [Nethercote and Seward 2007a] for storing the various meta data information we use for implementing SRS. We had to manually identify the starting instruction address, where processing of every user request commences. Once we encounter this address, we allocate the TrackLog and increment the current request id. With Valgrind, we were able to catch the out-of-bounds OS exception, and then continue execution in suppression mode. After completing the faulty user request and encountering the start address of the next request, the execution leaves suppression mode and enters recovery mode, where every load includes the fail safety check. However, since the Valgrind infrastructure, which is built for the ease of writing complex tools, imposes higher overheads, we also implemented a performance optimized version in *dynamoRIO* [Bruening et al. 2003]. This version of SRS, which we use to measure the overhead of SRS, includes all instrumentation of the prior version, but does not catch OS exceptions, so it has to be manually run in normal mode or recovery mode. All performance results reported in this paper are the overheads obtained with dynamoRIO tool.

### 4.2 Recovery in the presence of faults

We used versions of the server programs with real memory errors. The bugs in the program are described in the Table. 5.

For each of the programs, we connected the server with about 10 user requests, with the special user request that triggers the fault as the 5th request. We also used different user requests compared to the one used in profiling runs. In each of the cases, the bug causes an out-of-bounds OS exception which SRS catches; then, SRS enters suppression mode under which SRS was able to safely execute to the end of the faulty request, without experiencing other faults. Once the faulty request is "processed", SRS enters recovery mode in which fail safety checks are added before every load. In our set of experiments, we found that none of the fail safety checks failed; thus the need to restart the server never arises. We believe that typically, the need to restart will not occur, since the shared variables stay mostly the same irrespective of the variation in the user request. Thus this experiment shows that SRS can be used to survive faults safely.

### 4.3 Performance of SRS

In this experiment, we wanted to measure the overhead in the response time imposed by SRS during normal run as well as during recovery. Recall that the overhead imposed during normal run is due to the additional instrumentation involved for maintaining the TrackLog and for storing the current user request id for every original store instruction. As we can see from Fig. 13, the overhead imposed is very low, on an average, 5% across all benchmarks. This overhead is low mainly for two reasons. First, since these are not computationally bound programs, the additional instrumentation could easily be tolerated. Second, the additional instrumentation during normal run, is only an additional store for every store instruction. Since the processor does not generally wait for the stores, the overhead imposed is not high.

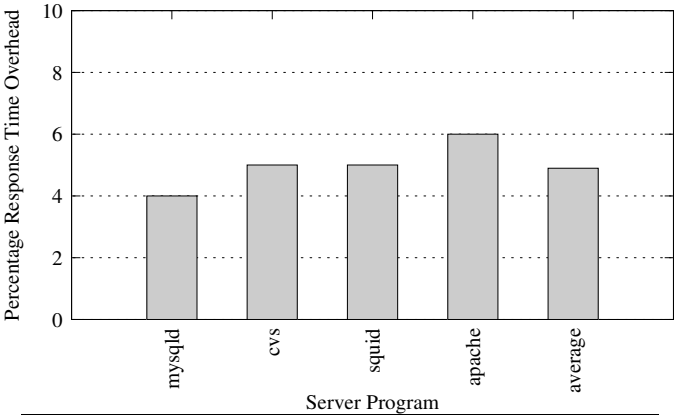


**Figure 13.** Response Time Overhead in Normal Run.

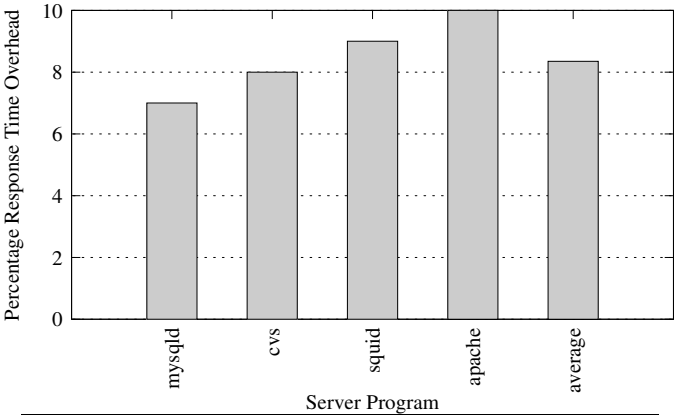


**Figure 14.** Response Time Overhead after Recovery.

We also measured the additional overhead imposed after recovery, which is basically the overhead for performing the fail safety checks along with every load. Even this overhead is pretty low, on an average 8% across each of the benchmarks (Fig. 14). The overhead is higher because now the instrumentation is performed for every load along with a safety check.

Finally, we also measured the overhead involved in executing in suppression mode. As we can see, from Fig. 15, this overhead can be as high as 3 times for these benchmarks. However it is important to note that the overhead for performing suppression is only for the duration of processing the *faulty request*, hence the dip in performance is only applicable for a very short time. Once the faulty request is handled, then *SRS stops executing in suppression mode*. Furthermore, there has been significant research on performing dynamic information flow tracking efficiently using hardware support [Suh et al. 2004, Crandall et al. 2006, Dalton et al. 2007]. DIFT hardware can be used to optimize the performance of the suppression mode, since the instrumentation operations performed during

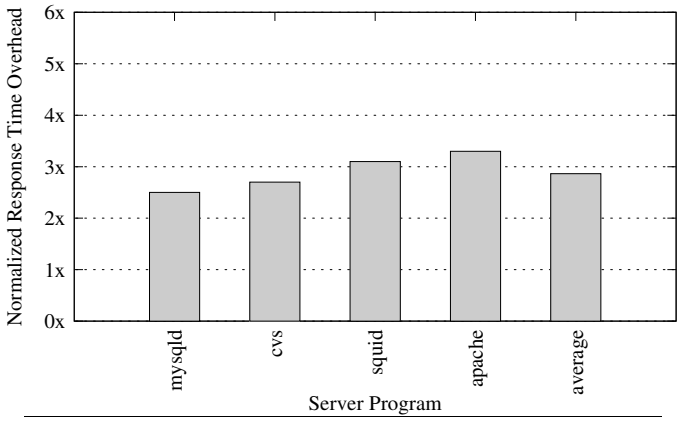


**Figure 15.** Response Time Overhead during Recovery in Suppression Mode.

suppression mode resemble those that are performed during dynamic information flow tracking.

**Performance of Checkpointing/Rollback Schemes.** One issue with Checkpointing based rollback recovery schemes is the frequency of checkpointing, which in turn results in the tradeoff between the normal execution performance and recovery performance. Infrequent checkpointing can reduce the overhead of checkpointing, but can increase the cost of recovery [Qin et al. 2005]. On the other hand, frequent checkpointing can cause greater overhead during normal execution. A highly optimized checkpointing system [Srinivasan et al. 2004] with a checkpointing interval of 50ms resulted in a overhead of about 11% during normal execution run, with marginal overheads during recovery. The overheads incurred by SRS are thus comparable (slightly lesser) to overheads experienced in a checkpointing based rollback recovery scheme, without the need for a complex checkpointing/rollback system.

## 5. Related Work

Recovering from failures has been a subject of significant research over the years. There has been significant work on coping with software failures by using various kinds of *rebooting* techniques. While whole program restart [Gray 1986] works by simply restarting the failed application, a small set of partial software components may be selectively restarted [Candea et al. 2004] to reduce the cost of recovery. The problem with restarting techniques is that the server program can be temporarily unavailable during restart. To reduce the cost of recovery, checkpointing based techniques[Gray 1986, Plank et al. 1998, Qin et al. 2005, Randell et al. 1978, Tallam et al. 2008], periodically checkpoint program state and rollback to the most recent checkpoint when the failure is detected. Having rollbacked to a safe state, that particular user request is then dropped [Qin et al. 2005, Tallam et al. 2008] so that the same failure is not repeated. However, the space and time overheads of checkpointing can be expensive if the checkpointing interval is small [Plank et al. 1998, Qin et al. 2005]. On the contrary, if the checkpoint interval is large then the throughput and the response time of the server can be affected during recovery.

There has also been work on the reliability of *long running programs* using a combination of checkpointing and tracing [**?**Zhang et al. 2006]. The scope of this work is to consider server programs, since there is a pressing need for them to be available. However, the techniques presented in this paper are also applicable for other long running programs, which process different user inputs continually.

There has been recent research on recovering from memory errors without the need for checkpointing or rollbacks [Rinard et al. 2004, Sidiroglou et al. 2005]. Our work is closely related to *failure oblivious computing* [Rinard et al. 2004], which also observes and utilizes the *self cleansing* property for maintaining server availability amidst failures. In the above work, instead of crashing when an illegal memory access occurs, the server continues program execution by simply discarding the illegal writes and manufacturing values to return for illegal reads. The success of the above technique hinges on small error propagation distances on server programs, which we refer to as *self cleansing* in our work. In our work, in addition to studying the *self cleansing* property in detail, we also use it in SRS differently compared to failure oblivious computing. Instead of speculating the programmer's intentions (for example, by manufacturing values for reads), in SRS we nullify the faulty request using *crash suppression* and isolate the faulty request from other requests.

Recovery oriented computing [Oppenheimer et al. 2002, Patterson 2002] proposes a system in which software components of a system are designed to be isolated, so that the impact on failures can be reduced. Our work is also related to work on recovering from failing device drivers [Swift et al. 2004, 2003] in that the above works also try to build a system that tries to isolate the failing device drivers from other parts of the system. In our work, we showed how this isolation is already present is server programs, to some extent. We then use this property to our advantage to build SRS, which is fail safe.

There has been significant work on recovering from *Transient soft errors* [Mukherjee et al. 2005, Reis et al. 2005, Reinhardt and Mukherjee 2000, Wang et al. 2007, Vijaykumar et al. 2002], which are radiation induced errors that cause random bit flips in both the computational and memory hardware. Bit flips to computation circuitry and memory elements are a form of memory corruption, and the results of the study conducted in this work are equally applicable for transient errors. In particular, the observation that a corrupted memory location , most often, corrupts only a few other memory locations can be taken advantage by a system that recovers from transient errors. Finally in this work, we merely used out-of-bounds memory accesses to trigger a fault, we can use dynamic security detection tools [Cheng et al. 2006, Newsome and Song 2005, Suh et al. 2004, Qin et al. 2006] or other assertion checks for this purpose.

## 6. Conclusions

In this paper, we conducted a detailed study of memory corruption propagation and isolation for a set of real world server programs. Our study shed light on an useful insight: the *self-cleansing* property which basically says that memory corruption does not generally propagate across user requests; on the contrary, it is cleansed automatically as memory gets deallocated or corrupt values are overwritten. We found out that this was because most of the values that are written during the processing of a user request are used locally; only a small (fixed) amount of global state is shared across user requests. Motivated by this insight, we proposed SRS, a technique that enables self recovery in server programs without requiring checkpointing or rollback. In SRS, we suppress the crash and execute forwards, using a demand driven approach to restore corrupted memory locations. Our experiments showed that SRS is able to recover from real world memory bugs in server programs; At the same time, SRS is very efficient causing only 5% overhead during normal run and 8% overhead after recovery.

## Acknowledgments

This work is supported by NSF grants CNS-0810906, CNS-0751961, CCF-0753470, and CNS-0751949 to the University of California, Riverside. We would like to thank the anonymous reviewers for providing useful comments to improve the paper.

## References

mysql bug. bugs.mysql.com/bug.php?id=110.

National vulnerability database. http://nvd.nist.gov/statistics.cfm.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO*, pages 265–275. IEEE Computer Society, 2003.

George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot - a technique for cheap recovery. In *OSDI*, pages 31–44, 2004.

Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. *ISCC*, pages 749–754, 2006.

JaeWoong Chung, Michael Dalton, Hari Kannan, and Christos Kozyrakis. Thread-safe binary translation using transactional memory. In *HPCA*, 2008.

Jedidiah R. Crandall, S. Felix Wu, and Frederic T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 3(4):359–389, 2006.

Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ISCA*, pages 482–493, 2007.

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

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: A benchmark for evaluating bug detection tools. In *Bugs*, 2005.

Shubhendu S. Mukherjee, Joel S. Emer, and Steven K. Reinhardt. The soft error problem: An architectural perspective. In *HPCA*, pages 243–247, 2005.

Vijay Nagarajan and Rajiv Gupta. Architectural support for shadow memory in multiprocessors. In *VEE*, pages 1–10, 2009.

Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *VEE*, pages 65–74, 2007a.

Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavy-weight dynamic binary instrumentation. In *PLDI*, pages 89–100, 2007b.

James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

David L. Oppenheimer, Aaron B. Brown, James Beck, Daniel Hettena, Jon Kuroda, Noah Treuhaft, David A. Patterson, and Katherine A. Yelick. Roc-1: Hardware support for recovery-oriented computing. *IEEE Trans. Computers*, 51(2):100–107, 2002.

David A. Patterson. Recovery oriented computing: A new research agenda for a new century. In *HPCA*, page 247, 2002.

James S. Plank, Kai Li, and Michael A. Puening. Diskless checkpointing. *IEEE Trans. Parallel Distrib. Syst.*, 9(10):972–986, 1998.

Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies - a safe method to survive software failures. In *SOSP*, pages 235–248, 2005.

Feng Qin, Cheng Wang, Zhenmin Li, Ho seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *MICRO 39*, pages 135–148, 2006.

Brian Randell, P. A. Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Comput. Surv.*, 10(2):123–165, 1978.

Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, pages 25–36, 2000.

George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *CGO*, pages 243–254, 2005.

Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference, General Track*, pages 149–161, 2005.

Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *ATEC*, pages 29–44, 2004.

G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, pages 85–96, 2004.

Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers (awarded best paper!). In *OSDI*, pages 1–16, 2004.

Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *SOSP*, pages 207–222, 2003.

Sriraman Tallam, Chen Tian, Rajiv Gupta, and Xiangyu Zhang. Avoiding program failures through safe execution perturbations. In *COMPSAC*, pages 152–159, 2008.

T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *ISCA*, pages 87–98, 2002.

Cheng Wang, Ho-Seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *CGO*, pages 244–258, 2007.

Xiangyu Zhang, Sriraman Tallam, and Rajiv Gupta. Dynamic slicing long running programs through execution fast forwarding. In *SIGSOFT '06/FSE-14*, pages 81–91, 2006.