Diverse Partial Memory Replication

Ryan M. Lefever, Vikram S. Adve, and William H. Sanders
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{lefever,vadve,whs}@illinois.edu

Abstract

An important approach for software dependability is the use of diversity to detect and/or tolerate errors. We develop and evaluate an approach for automated program diversity called Diverse Partial Memory Replication (DPMR), aimed at detecting memory safety errors. DPMR is an automatic compiler transformation that replicates some subset of an executable's data memory and applies one or more diversity transformations to the replica. DPMR can detect any kind of memory safety errors in any part of a program's data memory. Moreover, DPMR is novel because it uses partial replication within a single address space, replicating (and comparing) only a subset of a program's memory. We also perform a detailed study of the diversity mechanisms and state comparison policies in DPMR (a first of its kind for such diversity approaches), which is valuable for exploiting the high flexibility of DPMR.

Keywords: software memory errors, diversity, replication, fault injection, experimental evaluation

1. Introduction

In recent years, software dependability has become a major concern. Notable software failures have cost individual companies millions, and in some cases billions, of dollars [8]. A 2002 study by the National Institute of Standards and Technology (NIST) estimated that software errors cost the U.S. nearly 60 billion dollars annually [15]. Another study estimated that 40% of unplanned application downtime can be attributed to application failures [22].

Memory errors in unsafe languages like C and C++ are a notorious source of errors. Such mismanagement can lead to buffer overflows, reads and writes after free, frees after free, and uninitialized reads. Memory errors are difficult to handle because they are often deterministically activated and are important because they are often an easy target for attackers. A *deterministically activated fault* is a fault F in a program P, defined such that if F can manifest as an error when input I is applied to P, then F will always manifest as an error when I is applied to P. Traditional dependability techniques such as process replication, checkpoint-rollback, and reboot cannot handle deterministically activated faults. Those techniques rely on multiple executions of the same piece of code. Unfortunately, deterministically activated faults manifest the same way in every execution, rendering those techniques ineffective.

An important approach for dealing with deterministically activated faults is the use of diversity. This approach uses *differing executions* (of either a single program with different environmental parameters, or of different but equivalent programs) to detect and/or recover from errors. Diversity has two important advantages as an approach for software dependability. First, it can handle deterministically activated faults, because a deterministically activated fault will produce different results in different executions. Second, with a single strategy, diversity can handle *arbitrary* kinds of faults, including memory faults and (for some diversity mechanisms) logical faults.

1.1 Related Work

There has been a lot of dependability research addressing software memory errors. Traditional techniques for handling software memory errors utilize runtime safety checks. Although early work focused on spatial errors, recent work has included temporal errors. That is the case with Purify [12], Valgrind [23], CCured [14], SAFECode [11], and the smart pointer techniques in [3, 27].

More recent work has embraced the use of software diversity. For example, DieHard [5] diversifies heap memory through randomization and over-provisioning to achieve fault avoidance. Exterminator [16], utilizing the DieHard heap allocator, combines heap diversification and either process replication or an accumulation of executions to detect and patch software memory errors. Rx [18] makes use of diversified replay to recover from errors in a checkpoint-rollback system. Finally, Samurai [17] replicates heap memory that has been designated as critical by the programmer,

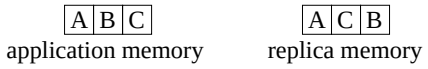


Figure 1. Diversity Example

in combination with the diversity provided by the DieHard heap allocator, to achieve fault detection and in some cases recovery. Unfortunately, the approaches mentioned above suffer from at least one of the following: lack of tunability, required human involvement, a restricted fault model, large overheads, or an inability to be used with error correction.

Diversity approaches can be broadly classified as either *diverse redundant software* (i.e., executions of two or more different but equivalent versions of a program) or *diverse software execution* (i.e., different behaviors of the same program in different executions, via different environmental parameters). Our work falls into the category of diverse redundant software, as do Samurai, Orchestra, and Exterminator under process replication. Other examples of diverse redundant software, which are not specifically targeted at software memory errors, include various N-Version software systems [7, 9, 10, 21, 25] and Recovery Blocks [20]. On the other hand, DieHard, Rx, and Exterminator without replication are examples of diverse software execution. Diverse software execution also includes dependability techniques such as randomized instruction sets [4, 13], randomized address spaces [6, 26], and portable checkpointing [19, 24].

1.2 Our Approach

In this paper, we develop an approach for automated program diversity called *Diverse Partial Memory Replication* (DPMR). DPMR is an automatic compiler transformation that replicates some subset of an executable's data memory and applies one or more transformations (which we call "diversity mechanisms") to the replica. The goal of the diversity is to cause memory errors to manifest differently in application memory and replica memory. For example, consider the memory layouts in Figure 1. If a buffer overflow were to spill out of object *A*, it would corrupt object *B* in application memory and object *C* in replica memory. If the overflow had a different value from that originally stored in object *B*, then the overflow could be detected through comparison of the values of *B* in application memory and replica memory.

Unlike traditional replication approaches, DPMR utilizes a *partial replication* approach that limits replication to an application's memory subsystem. By doing so, DPMR eliminates replication of computations that are not stored to memory. The results of those computations are stored in both the original memory and the replica. To make that efficient and correct, DPMR combines both the original behavior and the replica in a single process. This is a key difference from previous systems, which generally use separate OS processes for the original and the replica. An important challenge for such a partial replication approach is how to handle pointers that are stored in the two versions. Solving this issue is a key technical contribution of this paper, and is described in Section 2.

Another important aspect of DPMR is flexibility. A second goal of this paper is to study that flexibility. In particular, the study analyzes how different diversity mechanisms and state comparison policies affect the performance and dependability attributes of DPMR. The study is important because different applications may perform better under different diversity mechanisms and state comparison policies. Furthermore, different deployment environments may have different performance and dependability requirements. For example, it may be desirable to spend more resources on dependability when a new version of an application is deployed than when an older-trusted version is deployed.

The major contributions of this paper are as follows:

- We introduce an automated and flexible methodology for detecting memory errors, called DPMR. It detects both spatial and temporal memory errors, in all forms of memory, including global variables, the stack, and the heap. DPMR can also detect memory hardware faults, although that is not the focus of this paper.
- We provide a first-of-its-kind study of diversity mechanisms and state comparison policies (for diverse redundant software that addresses memory errors). This study will be instrumental in fulfilling DPMR's goal of flexibility as well as fulfilling that goal for other systems that use diverse redundant software for software memory errors.
- We introduce the idea of partial replication, which improves replication overhead by automatically replicating only the software components that are relevant to the targeted fault model, and places a replica in the same process as the corresponding original component.[1] There are many advantages to intra-process replication, as discussed in Section 2. Our primary contributions regarding partial replication have been to show how to do it and to evaluate key design choices. Comparison with full replication is left to future work.

The remainder of this paper is organized as follows. Section 2 presents the DPMR approach. Section 3 discusses the experimental framework and measures used to study DPMR. Sections 4 and 5 analyze results for different DPMR diversity mechanisms and comparison policies, respectively. Finally, Section 6 concludes the paper.

[1] The automatically chosen, intra-process replication presented here is similar in concept to the programmer-specified replication used in Samurai [17].

2. Diverse Partial Memory Replication

The replication strategy for DPMR is unique among replication strategies. While most software replication utilizes process replication, DPMR follows a partial replication strategy, in which only an application's memory subsystem is replicated. Such a partial replication strategy is achieved by performing memory replication in the same process as the original application. There are several advantages to such a strategy. First, instructions that do not pertain to the memory subsystem are not replicated, reducing overall computation. Second, it eliminates costly state transfer that would otherwise be required to share unreplicated computation, perform state comparisons, and distribute application input. Third, when an application interacts with its environment, such as with system calls, I/O, and signals, intra-process replication eliminates the need for replica synchronization. Fourth, intra-process replication eliminates the OS and library support that would be required to replicate system interactions. Finally, intra-process replication creates implicit diversity, as discussed in Section 4.1.

Error detection is achieved by comparing memory values loaded from application objects with memory values loaded from replica objects. An important issue arises with such detection. It concerns the problem of how to handle pointers that are stored in memory. There are two possibilities. The first is to store the same pointers to replica objects that are stored in application objects. The problem with that approach is that when the application traverses pointer-based data structures in memory, there is no way to traverse the corresponding replica data structures, because the requisite replica pointers are not stored in replica objects. The second possibility is to have replica data structures mirror application data structures. In this option, the pointers stored to replica objects are pointers to other replica objects, not pointers to application objects. The down side is that when pointers are loaded from memory, they are not comparable, potentially reducing error detection. DPMR uses the first option, in which replica objects store application pointers, so as not to suffer a loss of error detection.

2.1. Shadow Data Structures

As discussed above, when replica objects store application pointers, there is no way for the traversal of pointer-based replica data structures to mirror the traversal of the corresponding application data structures. In order to overcome that shortcoming, DPMR uses shadow data structures that mirror the structure of application data structures and maintain pointers to replica objects that facilitate the traversal of replica objects.

The shadow data structure concept is illustrated with a linked list in Figure 2. Each node in the linked list holds a

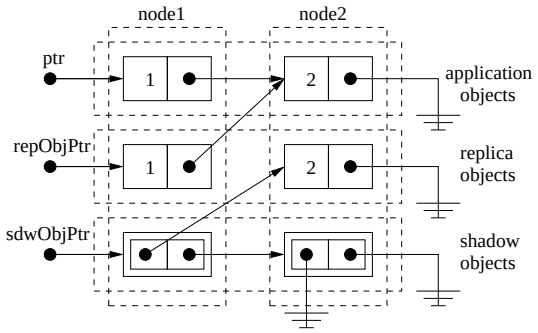


Figure 2. Linked List Memory Organization

data value and a pointer to the next node in the linked list. Application nodes occupy the top dashed box in the figure. Replica objects contain exactly the same values (including pointers) as the original application objects. Therefore, the *node*1 replica contains the data value 1 and a pointer to the *node*2 application object. Likewise, the *node*2 replica contains the data value 2 and a null pointer. Since the *node*1 application object points to the *node*2 application object, the *node*1 shadow object points to the *node*2 replica object. That provides a way to get to the replica when the application traverses the pointer in *node*1. There also needs to be a way to get to the next shadow object during such a traversal. Therefore, for every replica pointer r in a shadow object, there is also a shadow object pointer corresponding to r. In the figure, *node*1's shadow object has a pointer to *node*2's shadow object, to correspond to the *node*2 replica pointer.

The shadow data structure type corresponding to an application object with type s is defined by $\mathbf{st}(s)$ in Equation 1.

$$\mathbf{st}(s) = \begin{cases} \mathbf{st}(t)[] & \text{if } s = t[] \\ \text{struct } \{\mathbf{st}(t_0);...;\mathbf{st}(t_n);\} & \text{if } s = \\ & \quad \text{struct } \{t_0;...;t_n;\} \\ \text{union } \{\mathbf{st}(t_0);...;\mathbf{st}(t_n);\} & \text{if } s = \\ & \quad \text{union } \{t_0;...;t_n;\} \\ \text{struct } \{t*;\mathbf{st}(t)*;\} & \text{if } s = t* \wedge \mathbf{st}(t) \neq \text{ø} \\ \text{struct } \{t*;void*;\} & \text{if } s = t* \wedge \mathbf{st}(t) = \text{ø} \\ \text{ø} & \text{else} \end{cases} \quad (1)$$

Most of the equation can be summarized in three rules. 1) If the input type is a container type, such as an array, struct, or union, then the shadow type is that kind of container applied to the shadow types of the container's elements. 2) If the input type is a pointer $t*$, then the shadow type is a structure containing $t*$ (to point to a replica) and a pointer to t's shadow type (to point to the next shadow object). 3) If the input type is any other type, then the shadow type is null. There is a small exception to rule 2. If the shadow type of t is null, then instead of the shadow type of $t*$ be-

ing a one-element structure, a *void*$*$ type is added to the structure, in the place where a shadow object pointer would normally reside.

2.2. Transformation for DPMR

The approach here uses compiler transformations to automatically adapt application source code for DPMR. The transformations are implemented using the LLVM compiler framework [1]. An important aspect of using LLVM is that DPMR can be applied to any application written in an unsafe programming language for which there is an LLVM frontend. Below, we discuss the transformation process.

- *Memory allocation and deallocation:* When memory is allocated for an application object, whether it be in global variables, the stack, or the heap, memory is also allocated for a replica and a shadow object, in the same memory space. If an object O of type t is allocated and $\mathbf{st}(t)$ is null, then no shadow object is allocated. The shadow object pointer corresponding to a pointer to O will be null. When memory is deallocated, replica objects and shadow objects are also deallocated. Since there may not be a shadow object to deallocate, a null pointer check must be performed before a shadow object is deallocated.

- *Stores:* Memory stores are broken into two cases: storage of non-pointers, and storage of pointers. When the original application stores non-pointer data to an object, the same data are stored to the corresponding replica object. Storage of pointers is a bit more complicated. Let's say pointer P is stored to object O, with corresponding replica object O_r and shadow object O_s. In addition, let P_r and P_s point to the replica and shadow objects corresponding to the object pointed to by P, respectively. As with non-pointer data, the data stored to the original application object are also stored to the replica object. Thus, P is stored to both O and O_r. On the other hand, O_s is updated with P_r and P_s.

- *Loads:* Loads are transformed in two ways. First, loads are replicated for error detection purposes. Therefore, a value loaded from an application object is compared to a value loaded from a replica object. Second, if the value loaded from an application object O is a pointer P, regardless of whether replica memory is loaded for detection purposes, a replica object pointer and shadow object pointer corresponding to P must also be loaded from O's shadow object.

- *Function calls:* Function types and function calls must be augmented to allow replica and shadow object pointers to be passed and returned along with pointer arguments and pointer return values. Function types are modified according to the augmented function type below.

$$\mathbf{aft}(t_r(t_0,...,t_n)) = t_r(\mathbf{st}(t_r)*,\mathbf{st}(t_0),...,\mathbf{st}(t_n),t_0,...,t_n)$$

  An augmented function A has the same type as the function F on which it is based, with a few additional parameters at the beginning of its parameter list. The first new parameter is a shadow data pointer corresponding to F's return value. Space for $\mathbf{st}(t_r)$ is allocated on the stack of a calling function. $\mathbf{st}(t_r)*$ is followed by shadow objects $\mathbf{st}(t_i)$, corresponding to F's original parameters.

- *Pointer arithmetic:* Whenever pointer arithmetic is applied to an application pointer, similar arithmetic must be applied to the corresponding replica pointer and shadow object pointer. Such arithmetic includes casts.

2.3. External Function Wrappers

A common problem with code transformations is that it is not always possible or desirable to transform all of the code that will execute when an application is run. A common case of that is library code. In DPMR, that limitation is overcome with external function wrappers. By knowing the library function's memory access patterns with respect to memory accessible from the application, it is possible to write a wrapper that calls the original library function, updates replicas and shadow objects, and performs desired error-detecting comparisons. In that context, memory accessible from the application consists of memory accessible through arguments passed to a library function, the library function's return value, and library-exported global variables, such as stderr, stdout, and errno in C. It is not necessary to worry about memory that is completely internal to a library function, although neglecting it could potentially reduce error detection. In general, external function wrappers are easy to write, and our plan is for wrappers to be written for all common library functions.

2.4. Limitations

Unfortunately, the design presented here is not without its limitations. First, memory allocations must be typed correctly so that shadow objects can be allocated correctly. Second, loads and stores of pointers must be typed correctly so that corresponding shadow objects can be updated and loaded correctly. Third, int-to-pointer casts cannot be handled. Finally, pointer arithmetic must be typed correctly to allow corresponding pointer arithmetic to be applied to shadow object pointers.

Those limitations can be overcome, and DPMR can be designed to work with any arbitrary C/C++ program by

eliminating shadow data structures and having replica data structures mirror application data structures. Two insights allow such a design to work. First, if a pointer is stored to memory, but it is not known at compile time to be a pointer, then replica data structures will end up pointing into application data structures. Comparisons resulting from such decayed replica data structures will not produce relevant information; however, they will not result in false error reports either. Second, load comparisons are not required. Therefore, static analysis can eliminate load comparisons that might involve pointers.

We evaluated shadow data structures because, when applicable, they may be the preferred choice, since they cover pointer as well as non-pointer values. The alternative approach can handle the same error model, but the coverage is unclear. In practice, we expect that the two approaches can be combined using a compiler transform called "Automatic Pool Allocation," which is a focus of future work.

3. Experimental Framework

In this section we present the experimental framework that is used to evaluate DPMR. The study is conducted using non-fault injection experiments to evaluate DPMR overhead and fault injection experiments to evaluate its dependability properties. In order to compare different diversity mechanisms and comparison policies, under both fault injection and non-fault injection scenarios, an application is compiled into multiple variants. There are four classes of variants. The first class consists of one variant that has not been transformed with DPMR or instrumented with fault injection. It is referred to as the *golden* case. The second class also consists of one variant. That variant has not been transformed with DPMR but has been instrumented for fault injection. It is referred to as the *fi-stdapp* case. The third class of variants make up the *nofi-dpmr* case. Variants in that class are transformed with DPMR but not instrumented for fault injection. The final class, *fi-dpmr*, is transformed with DPMR and instrumented for fault injection.

Experiments were conducted using the `art`, `bzip2` 2, `equake`, and `mcf` applications from the SPEC CPU2000 benchmark suite [2]. All experiments were conducted on a homogeneous testbed with machines that have 2 GHz AMD Athlon processors, 512 MB of memory, and 256 KB L2 caches, and are running Ubuntu 8.10 Linux. All timing measures taken refer to wall clock time. Timeouts of approximately 20 times the normal application running time (under the current workload) are used in fault injection experiments. If an experiment exceeds the timeout value, the application is killed and considered to have produced incorrect output.

Notation	
$P_s[X]$	sample density function for the random variable X
$\mu[X]$	sample mean of X
Experiment Descriptors	
C	current comparison policy
D	current diversity mechanism
F	true for a fault injection experiment
Random Variables	
T	running time of the current experiment
SF	true for a successful fault injection
CO	true if the experiment produces the correct output
$Ndet$	true if a fault is naturally detected
$Ddet$	true if a fault is detected by DPMR
$T2D$	time to fault detection
$StdNotAllDet$	true if $P_s[(Ndet \mid C = \emptyset, D = \emptyset, F, SF, \neg CO) = 1] < 1$

Table 1. Measurement Components

3.1. Fault Injections

In this paper, faults are injected using a compiler-based framework that is designed to simulate software bugs. Faults are injected statically into application code to create a deterministically activated fault. That strategy is unlike runtime strategies that inject a fault once, rather than every time a targeted code executes. Two types of fault injections are used for this paper: heap array resizes, and immediate frees. A *heap array resize* reduces the number of objects requested in a heap array allocation. An *immediate free* deallocates heap memory as soon as it is allocated. The two types of fault injections were chosen because they result in the types of memory errors that are within the scope of DPMR. In particular, malloc array resizes lead to buffer overflows, and immediate frees lead to reads, writes, and frees after free. Separate experiments were conducted for each heap array resize and each immediate free. Heap array resizes were applied to all available heap array allocation sites, and immediate frees were applied to all heap allocation sites. In total, 64 heap array resizes and 55 immediate frees were injected.

It is important to note that not all fault injections manifest as errors. For one thing, the code containing an injection may reside in a part of the application that is not exercised by tested workloads. It is also possible that a fault is executed but does not manifest as an error. For example, consider a heap array resize in which the initial heap request is for 24 bytes, and the resize reduces it to a 16-byte request. It is common for `malloc` implementations to have a minimum allocation size, which might allocate 24 bytes for the 16-byte request. Static analysis was used to filter out parameters for fault injections that would definitely not manifest, but that determination is not always possible.

3.2. Measures

This section describes the measurements computed for experiments. An experiment is one run of an application variant. Each experiment is identified by the tuple $\{W, C, D, I, RN\}$. W is the workload. C is the comparison policy applied to the variant if the variant is transformed with DPMR. A comparison policy of ø indicates that DPMR was not used. D is the explicit diversity mechanism applied to the variant's replica memory. ø indicates that no explicit diversity is applied. I indicates the type of fault injection, and *RN* indicates the run number under the settings $\{W, C, D, I\}$.

Table 1 describes sub-components of measure definitions. Most items in the table are self-explanatory, but a few require further explanation.

- *successful fault injection:* A fault is considered successfully injected if the corresponding fault injection code is executed at least once during an experiment. A successful fault injection does not imply manifestation. Without intrusive instrumentation, error manifestation cannot be precisely determined. Therefore, it is not computed.

- *correct output:* Correct output indicates that an experiment produced the output that the golden run would have produced under the same workload. Incorrect output includes not only "bad" results but timeouts and error detection as well.

- *natural detection:* Natural error detection is error detection that is implicit to the application under study and can occur in two ways. An error is naturally detected if the variant in an experiment crashes. An error is also naturally detected if the variant produces application-dependent output indicating that an error was detected, such as an error message written to stderr or an exit with an error-identifying return value.

- *time to fault detection:* Time to fault detection measures the amount of time between the injection of a fault and the detection of an error. The time to error detection is not computed, because there is not a non-intrusive way to determine when a fault has manifested as an error.

- *StdNotAllDet*: *StdNotAllDet* is true for a given fault injection if, under the fi-stdapp case, there was at least one run that produced incorrect output but not natural error detection.

Below are the measurements used to evaluate DPMR.

- *overhead:* Overhead is defined as the ratio of a variant's non-fault injection running time to the golden case's running time. More formally:

$$\mu[T|C = c, D = d, \neg F] / \mu[T|C = ø, D = ø, \neg F]$$

- *coverage:* A fault is considered covered in successful fault injection experiments if the tested variant produces correct output, or the fault is detected. It is formally defined as

$$\mu[CO \vee Ndet \vee Ddet | C = c, D = d, F, SF]$$

- *conditional coverage:* Conditional coverage is coverage conditioned on incorrect output and *StdNotAllDet*. It is important because it measures how a variant would perform in cases where $fi\text{-}stdapp$ would have produced incorrect output but would not always have detected an error.

- *detection latency:* Detection latency is defined for covered experiments that exhibit some sort of detection. Formally, detection latency is

$$\mu[T2D | C = c, D = d, F, SF, \neg CO, (Ndet \vee Ddet)]$$

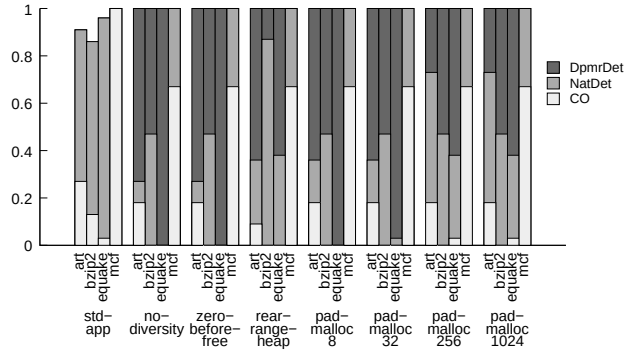
4. Study of Diversity Mechanisms

This section introduces several diversity mechanisms and evaluates them under a common comparison policy in which state is compared every time an application load occurs. Diversity is applied only to replica memory.

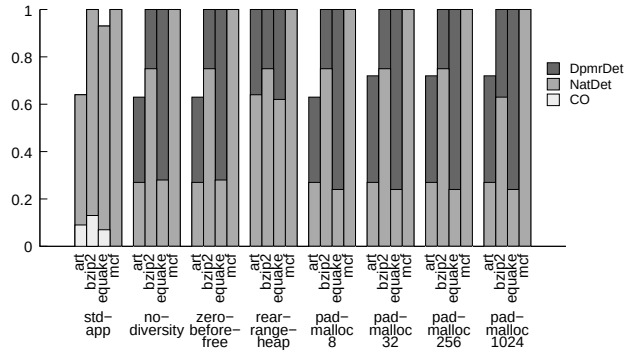
4.1. Diversity Mechanisms

Although explicit diversity is utilized by DPMR, there is already a natural diversity between application and replica objects due to the nature of intra-process replication. For example, if an object O, its replica O_r, and its shadow object O_s are allocated on the heap, they may very well be laid out consecutively. Thus, the object immediately following and immediately preceding O will be different from the objects immediately following and immediately preceding O_r.

The natural diversity inherent to intra-process replication makes up the first diversity mechanism that is evaluated. It is referred to as the *no-diversity mechanism*, implying that no explicit diversity is utilized. The second diversity mechanism is called *zero-before-free*. In variants using zero-before-free, whenever replica memory is deallocated, its contents are overwritten with a value of 0. Zero-before-free was chosen because it is likely to cause an object and its replica to differ should a dangling pointer be loaded. The third diversity mechanism is the *rearrange-heap* mechanism. It works as follows. Prior to allocation of a replica



(a) Heap Array Resizes



(b) Immediate Frees

Figure 3. Coverage of Diversity Mechanisms

object of size *s* on the heap, a random number *x* between 1 and 20 is chosen. The replica object is given the heap location that would normally be given if *x* objects of size *s* had been allocated on the heap immediately prior to the current request. The rearrange-heap mechanism was chosen because its random behavior decreases the likelihood that an application object and its replica will occupy the same pair of memory locations as a previous application object-replica pair, improving the detection of dangling pointers. The final diversity mechanism is *malloc padding*. When malloc padding is used, the size of a heap allocation request for a replica object is increased by a fixed padding amount. Malloc padding was used specifically to combat overflows. Because of the padding, the object immediately following a replica becomes less corrupted than the object immediately following the replica's corresponding application object.

4.2. Diversity Mechanism Results

We begin our evaluation of diversity mechanisms by examining coverage (Figure 3) and conditional coverage (Figure 4). Coverage results are broken into correct output, natural detection, and DPMR detection, while conditional coverage, which is already conditioned on incorrect output, is broken into natural detection and DPMR detection. We draw several insights from the two sets of coverage results. First, coverage numbers are high in general, which makes sense because many faults result in natural detection, such as a crash. Second, all of the buffer overflows resulting from heap array resizes were covered by all of the diversity mechanisms, suggesting that implicit diversity, due to intra-process replication, may be enough to detect buffer overflows. The implicit diversity makes it unlikely that the objects following an object *O* and *O*'s replica will be paired, causing buffer overflows to manifest differently in application memory and replica memory. Third, we observe that the rearrange-heap mechanism performs the best (in terms of coverage) and is the only diversity mechanism to cover 100% of the injected faults. Rearrange-heap performs well against heap array resizes for the same reasons that all of the diversity mechanisms perform well. It performs well against the dangling pointers created by immediate frees because, as mentioned above, its randomization decreases the likelihood that an application object-replica pair will occupy the same memory locations as previous application object-replica pairs. Our final observation is that 32 bytes of malloc padding offered the highest DPMR detection. Though it has not yet been investigated, it would seem reasonable that DPMR detection would tend to result in better fault diagnosis than natural detection would, because it would offer an opportunity to automatically log process state at the time of detection.

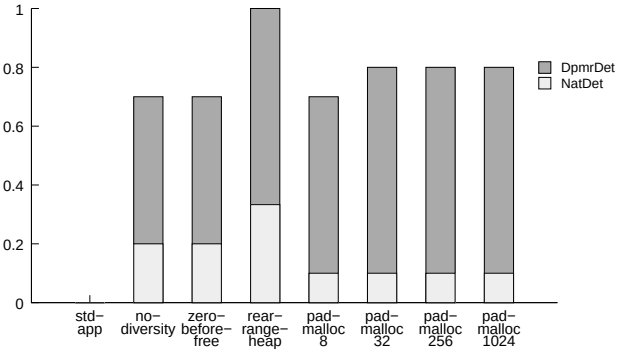


Figure 4. Conditional Coverage of Diversity Mechanisms

We now turn our attention to overhead. Overhead is displayed in Figure 5. We observe that all overheads are between 2x and 5x, with no-diversity and zero-before-free performing the best, and the larger pad-malloc policies performing the worst. We hypothesize that the rearrange-heap and pad-malloc mechanisms perform worse than the no-diversity and zero-before-free mechanisms because they may cause the heap allocator to cross cache page boundaries when allocating and deallocating buffers, with the larger

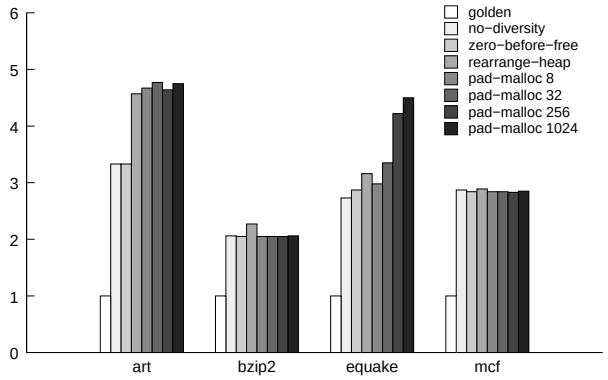


Figure 5. Overhead of Diversity Mechanisms

variant	msecs				
	art	bzip2	equake	mcf	
no-diversity	26	190	144	188065	heap array resizes
zero-before-free	18	182	142	191797	
rearrange-heap	21	124	178	494	
pad-malloc 8	18	189	146	187528	
pad-malloc 32	16	182	160	187410	
pad-malloc 256	21	182	270	189091	
pad-malloc 1024	46	182	639	189193	
no-diversity	7207	4584	154	9	immediate frees
zero-before-free	7275	4502	149	94	
rearrange-heap	29	4960	258	7	
pad-malloc 8	7404	4535	174	5	
pad-malloc 32	6742	4510	188	6	
pad-malloc 256	6430	4448	310	6	
pad-malloc 1024	6739	4526	722	7	

Table 2. Detection Latency of Diversity Mechanisms

pad-mallocs being the biggest offenders.

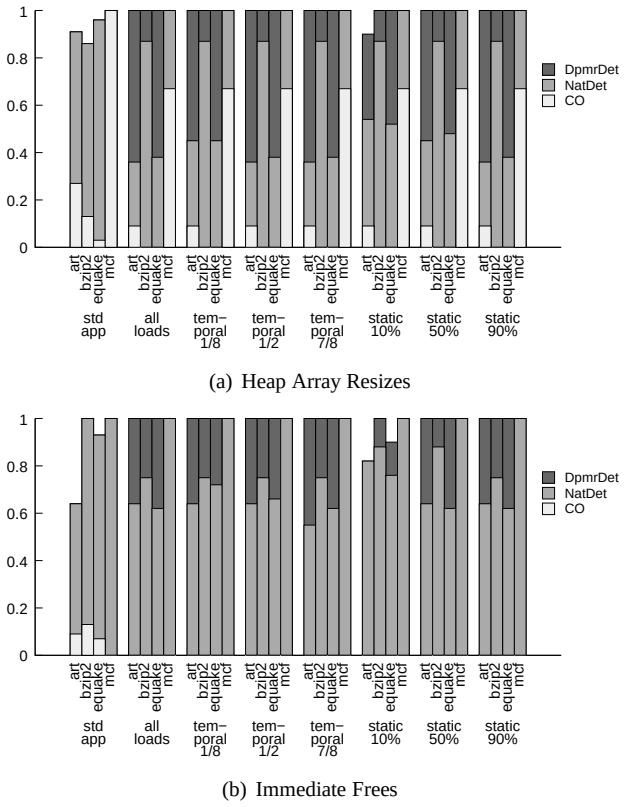
The last metric we examine is detection latency, shown in Table 2. Detection latency is important because the sooner an executed fault is detected, the sooner recovery can start, and the easier it will be to diagnose the fault. Rearrange-heap tends to have the best detection latency, given its drastic reduction in detection latency, compared to the other diversity mechanisms, for `mcf` heap array resizes and `art` immediate frees.

5. Study of Comparison Policies

In this section we evaluate several state comparison policies. They are evaluated using the rearrange-heap diversity mechanism presented in Section 4.

5.1. Comparison Policies

The first comparison policy is the *all loads* policy that was used in Section 4. Under the all loads policy, comparisons are made whenever a value is loaded from application memory. The second comparison policy is *temporal load-checking*. Under temporal load-checking, a temporal fraction of application loads are replicated and compared, to detect errors. Temporal load-checking is performed by keeping a counter and branching to comparisons when desired. The third comparison policy is *static load-checking*, in which the compiler statically replicates and adds comparison code at a fraction of application loads, chosen randomly. The temporal and static load-checking policies were chosen because of their potential to reduce overhead. Moreover, they can be tuned to meet the performance and dependability requirements of different deployment scenarios.



(a) Heap Array Resizes

(b) Immediate Frees

Figure 6. Coverage of Comparison Policies

5.2. Comparison Policy Results

As with diversity mechanisms, we start our evaluation of comparison policies by examining coverage (Figure 6) and conditional coverage (Figure 7). Our primary observation is that coverage is robust in the face of reduced checking. In fact, the only case in which coverage was reduced was when only a static 10% of application loads were checked. Those findings suggest that memory faults propagate to a large fraction of the tested programs. We believe that makes

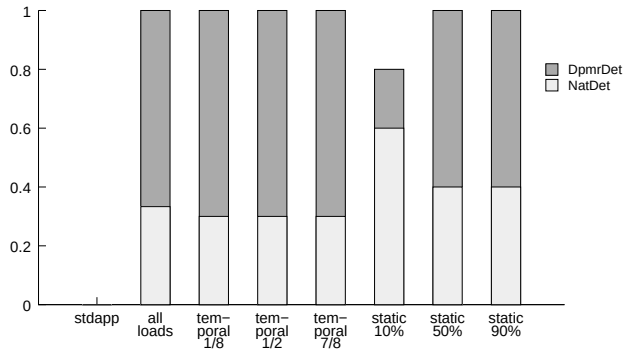


Figure 7. Conditional Coverage of Comparison Policies

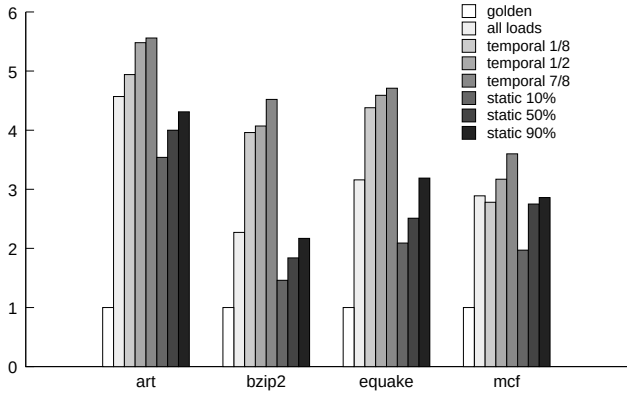


Figure 8. Overhead of Comparison Policies

sense, since it is likely that the data throughout an application are inter-dependent.

Intuitively, it would seem that the primary benefit of reduced-checking policies is a reduction in overhead. As seen in Figure 8, that was the case with static load-checking, culminating in the static 10% policy achieving a $1/3$ speedup. However, it was not the case when load comparisons were temporally reduced.

To understand why temporal checking increased overhead, we must understand that it requires an additional check to be made at each load, to determine whether the load's value should be verified. That leads to three additional sources of overhead. First, it inserts additional computation. Second, some of that additional computation comprises branches that can lead to expensive branch misses. Finally, we conjecture that additional branches may prevent standard compiler optimizations from occurring that would otherwise have taken place. We do not feel that temporal load-checking is necessarily a lost cause, but it would require aggressive optimizations to make it efficient. In particular, there are some compiler optimizations that could increase its performance if the load-checking were performed periodically.

variant	msecs				
	art	**bzip2**	**equake**	**mcf**	
all loads	21	124	178	494	heap array resizes
temporal 1/8	22	251	193	497	
temporal 1/2	21	258	184	501	
temporal 7/8	21	279	185	497	
static 10%	16	117	176	500	
static 50%	19	117	162	512	
static 90%	24	122	173	511	
all loads	29	4960	258	7	immediate frees
temporal 1/8	45	8630	336	3	
temporal 1/2	38	9032	331	4	
temporal 7/8	37	9884	338	3	
static 10%	15	3084	257	5	
static 50%	53	3754	242	7	
static 90%	29	4606	255	7	

Table 3. Detection Latency of Comparison Policies

Finally, we examine detection latency, shown in Table 3. The major finding is that detection latency tends to be lower for static load-checking than for the all loads policy, and it tends to be higher for temporal load-checking than for the all loads policy. Since static load-checking policies generally had lower overhead than the all loads policy and temporal load-checking policies generally had higher overheads, that may suggest that detection latency is dominated more by overhead than by checking frequency. Intuitively reduced overhead should reduce detection latency, and reduced checking should increase detection latency. However, reduced checking only negatively impacts detection latency between the time an error manifests and the time a detection occurs. On the other hand, reduced overhead positively impacts detection latency for a longer period of time, i.e., between the injection of a fault and detection.

6. Conclusion

In this paper, we introduced Diverse Partial Memory Replication (DPMR) for detecting spatial and temporal software memory errors that occur in any segment of application data memory. The DPMR approach is flexible and automated. It utilizes a partial replication framework that only replicates computation that is related to the targeted fault model, and places original computation and replica computation in the same process. To evaluate DPMR we conducted a first-of-its-kind study that compares different diversity mechanisms and state comparison policies. In that study, we found that different diversity mechanisms offer different properties with respect to coverage, overhead, and detection latency. In particular, we found that the rearrange-heap diversity mechanism was able to detect all injected

faults. We also found that coverage was robust in the face of reductions to state comparisons, and with a static 90% reduction in state comparisons, we were able to realize a speedup of approximately $1/3$.

Acknowledgments: The authors thank Jenny Applequist, Eric Rozier, and Saman A. Zonouz for their help editing this paper. This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-0406351 and 0615372. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] The LLVM compiler infrastructure. [Online]. Available: `http://www.llvm.org/`, Nov. 2009.

[2] SPEC CPU2000. [Online]. Available: `http://www.spec.org/cpu2000/`, Dec. 2009.

[3] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. *ACM SIGPLAN Notices*, 29(6):290–301, June 1994.

[4] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović. Randomized instruction set emulation. *ACM Trans. on Inform. and System Security*, 8(1):3–40, Feb. 2005.

[5] E. Berger and B. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 158–168, June 2006.

[6] S. Bhatkar, D. C. DuVarney, and R. Secar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. of the 12th USENIX Security Symp.*, pages 105–120, Aug. 2003.

[7] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Trans. on Comput. Syst.*, 21(3):236–269, Aug. 2003.

[8] R. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, Sept. 2005.

[9] L. Chen and A. Avižienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. of the 8th Int. Fault-Tolerant Computing Symp.*, pages 3–9, June 1978.

[10] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proc. of the 15th USENIX Security Symp.*, pages 105–120, Aug. 2006.

[11] D. Dhurjati, S. Kowshik, and V. Adve. SAFECode: Enforcing alias analysis for weakly typed languages. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 144–157, June 2006.

[12] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. of the Winter USENIX Conf.*, pages 125–136, Jan. 1992.

[13] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. of the 10th ACM Conf. on Comput. and Commun. Security*, pages 272–280, Oct. 2003.

[14] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. on Programming Languages and Syst.*, 27(3):477–526, May 2005.

[15] NIST: National Institute of Standards and Technology. Software errors cost U.S. economy $59.5 billion annually: NIST assesses technical needs of industry to improve software-testing. [Online]. Available: `http://www.nist.gov/public_affairs/releases/n02-10.htm`, June 2002.

[16] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 1–11, June 2007.

[17] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: Protecting critical data in unsafe languages. In *Proc. of the 3rd ACM European Conf. on Comput. Syst.*, pages 219–232, Apr. 2008.

[18] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies – A safe method to survive software failures. In *Proc. of the 20th ACM Symp. on Operating Syst. Principles*, pages 235–248, Oct. 2005.

[19] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogeneous architectures. In *Proc. of the 27th Int. Fault-Tolerant Computing Symp.*, pages 58–67, June 1997.

[20] B. Randell and J. Xu. The evolution of the recovery block concept. In M. R. Lyu, editor, *Software Fault Tolerance*, pages 1–22. Wiley, 1995.

[21] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of the 4th ACM European Conf. on Comput. Syst.*, pages 33–46, Apr. 2009.

[22] D. Scott. Making smart investments to reduce unplanned downtime. Tactical Guidelines TG-07-4033, Gartner Group, Mar. 1999.

[23] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proc. of the USENIX 2005 Tech. Conf.*, pages 17–30, Apr. 2005.

[24] K.-F. Ssu and W. K. Fuchs. PREACHES – Portable recovery and checkpointing in heterogeneous systems. In *Proc. of the 28th Int. Fault-Tolerant Computing Symp.*, pages 38–47, June 1998.

[25] K. S. Tso and A. Avižienis. Community error recovery in N-version software: A design study with experimentation. In *Proc. of the 17th Int. Fault-Tolerant Computing Symp.*, pages 127–135, July 1987.

[26] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proc. of the 22nd Int. Symp. on Reliable Distributed Syst.*, pages 260–269, Oct. 2003.

[27] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proc. of the 12th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 117–126, Nov. 2004.