A Mirrored Data Structures Approach to Diverse Partial Memory Replication

Ryan M. Lefever, Vikram S. Adve, and William H. Sanders
University of Illinois at Urbana-Champaign
1308 West Main Street
Urbana, IL 61801, USA
ryan.lefever@gmail.com, vadve@illinois.edu, whs@illinois.edu

***Abstract*—Software memory errors are a growing threat to software dependability. In previous work, we proposed an approach for detecting memory errors, called Diverse Partial Memory Replication (DPMR), that utilized automated program diversity and memory replication. The original design aimed to maximize coverage by making the pointers stored in different memory replicas comparable. In this paper, we propose and evaluate an alternative design called Mirrored Data Structures (MDS), which sacrifices pointer comparability to gain three primary benefits. 1) MDS significantly increases DPMR's applicability by eliminating all DPMR restrictions on memory allocation, pointer arithmetic, and pointer-to-pointer casts. 2) For programs that store many pointers to memory, MDS reduces DPMR's overhead, as is demonstrated in experimental results. 3) MDS significantly reduces DPMR's memory footprint.**

***Keywords*-software memory errors; diversity; replication; fault injection; experimental evaluation**

I. Introduction

Software dependability is a growing societal concern as computer systems become increasingly pervasive. Software failures not only result in significant monetary losses, such as those described in [1] and [2], but can also jeopardize human lives. A significant source of software errors is mismanaged memory in unsafe programming languages, such as C and C++.

Mismanaged memory can lead to buffer overflows, dangling pointers, and uninitialized reads. Like other software bugs, those errors can result in crashes and incorrect output. In addition, memory errors are notorious for opening vulnerabilities that attackers can use to compromise a system. Many buffer overflows have made their way into production systems and lend themselves to being exploited [3]. In more recent times, exploitable dangling pointers have shown up in applications such as the CVS revision control system [4], the MIT Kerberos authentication daemon [5], the Opera web browser [6], and the IIS web server [7].

Unfortunately, software memory errors have been difficult to detect, diagnose, and correct. Comprehensive static analysis is intractable, and practical static analysis and testing have proven to be insufficient for eliminating memory errors. One approach to deal with software memory errors is to combine software diversity and replicated execution. By replicating the execution of a software component and applying well-crafted diversity to one of the replicated components, we can cause faults to manifest differently in each replica.

In [8], we proposed an approach for automated program diversity, called *Diverse Partial Memory Replication (DPMR)*. DPMR is an automated compiler transformation that replicates a subset of a program's data memory and applies one or more diversity transformations to the replica. DPMR detects spatial and temporal software memory errors, as well as hardware memory errors, in all segments of application data memory, i.e., heap memory, stack memory, and global variables.

DPMR is powerful; however, the design in [8] does not work with many C programs, due to its restrictions on many non-type-safe constructs. DPMR also has high memory consumption, and fairly high performance overhead. Until these problems are resolved, the approach has limited practical value. This paper proposes an alternative design that sacrifices pointer comparability to solve the key practical problems listed above.

The approach to DPMR in [8] utilized a memory layout called *Shadow Data Structures (SDS)*. SDS stores the same pointer values to replica memory as are stored to application memory. The benefit of that approach, and the reason it was investigated first, is that it maximizes coverage by making all data in application memory comparable. When evaluated, SDS demonstrated very high levels of coverage, giving us confidence that an alternative memory layout could tolerate a small loss in coverage to greatly improve DPMR's applicability and performance.

In this paper, we propose the *Mirrored Data Structures (MDS)* memory layout. It sacrifices comparability of pointers stored in memory in order to gain the following benefits. 1) MDS significantly increases DPMR's applicability by eliminating all DPMR restrictions on memory allocation, pointer arithmetic, and pointer-to-pointer casts. 2) MDS provides performance improvements for pointer-heavy programs, as demonstrated in experimental results. 3) MDS significantly reduces the memory footprint of DPMR. Despite sacrificing pointer comparability, MDS detects the same class of faults as SDS.

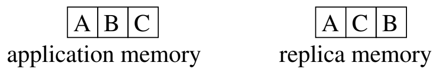


Figure 1: A Diverse Memory Layout

The purpose of this paper is to present and evaluate MDS and to compare it to SDS. In Section II, we review DPMR and the SDS approach. In Section III, we define MDS and compare it to SDS. In Section IV, we describe our experimental framework, and in Section V, we evaluate MDS using experimental results. Section VI presents related work. Finally, Section VII concludes this paper.

II. Diverse Partial Memory Replication

In this section, we review DPMR and the SDS memory layout from [8]. We focus on features that are relevant to a comparison of MDS and SDS. Tunable components that are directly applicable to both approaches, such as diversity transformations, are given cursory coverage here and are discussed in more detail in Section V.

A. Approach

DPMR is an automated compiler transformation that transforms a program in three ways. First, it replicates an application's memory subsystem. Second, it applies a diversity transformation to replica memory. Finally, when application memory is loaded, replica memory is loaded, and the results of the two loads are compared. If the loaded values differ, then an error has been detected.

To understand the premise behind DPMR, consider the memory layouts in Figure 1. On the left side, application objects are laid out such that B follows A and C follows B. In the replica layout on the right side, the order is diversified, with C following A and B following C. If a buffer overflow were to spill out of object A, object B would be corrupted in application memory, while object C would be corrupted in replica memory. The buffer overflow could be detected through comparison of the value of object B (or C) in application memory to the value of object B (or C) in replica memory.

Replication in DPMR is based on a partial replication strategy that is unique. Traditionally, software replication uses a process replication scheme in which an entire process is replicated. Our partial replication strategy only replicates an application's memory subsystem and does so in the same process as the original application behavior. Such intra-process partial replication eliminates many sources of overhead that are unavoidable in other process replication strategies. For example, it does not replicate instructions that do not pertain to the memory subsystem.

B. Shadow Data Structures

As mentioned above, DPMR detects errors by comparing values loaded from application memory and replica memory.

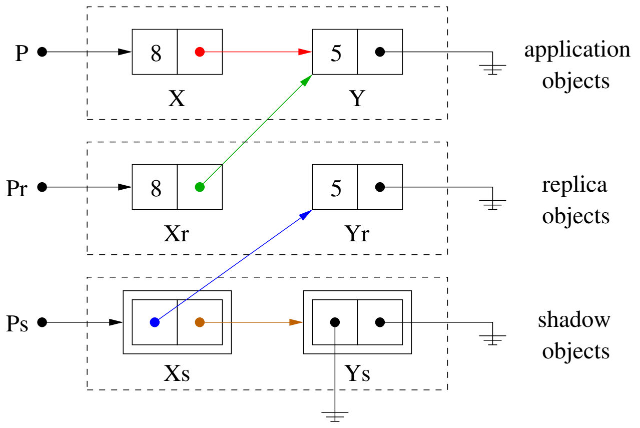


Figure 2: Shadow Data Structures Example

A key technical challenge is that of determining how to handle pointers that are stored in memory. In order to maximize coverage, SDS employs a strategy in which all values stored in application memory can be verified by values in replica memory. Such a strategy involves storing the same pointer values to replica memory as are stored in application memory.

Consider the top third of Figure 2.[1] There are two objects, X and Y, that reside in an application's memory. Object X contains a pointer to object Y. When the application is transformed by DPMR, replica objects in the middle third of the figure are created. Since X has a data value of 8 stored to it, so does X_r. Under the SDS memory layout, the pointer that is stored to X_r is identical to the pointer stored to X, making them comparable.

Unfortunately, storing the same pointer to application and replica memory causes a problem. In particular, if the application traverses the pointer stored in X in order to access Y, DPMR is left without a way to get to Y_r. SDS solves that problem. When SDS allocates an application object, it allocates a shadow object as well as a replica object. For every pointer stored to application memory, the same pointer is stored to replica memory, and a replica pointer is stored to shadow memory. In Figure 2, the first pointer in X_s points to Y's replica, since the pointer stored to X points to Y. Additionally, we need a way to get to Y's shadow object, when X's pointer to Y is traversed. Therefore, the second pointer stored in X_s points to Y_s.

Shadow objects are given shadow types as defined in Table I. The table can be broken into three logical rules. 1) The shadow types for composite types, such as arrays, structures, and unions, are defined in rows 1-3 of the table. The shadow type of a composite type C is the type that results from applying the same composite type to the shadow types of C's elements. Thus, an array with elements typed τ becomes an array with elements typed $\mathbf{st}(\tau)$. 2) The shadow

[1]The symbol comprising three horizontal bars indicates a null pointer.

Table I: Shadow Type Definition

Description	Original Type t	Shadow Type $\mathbf{st}(t)$
array	$\tau[]$	$\mathbf{st}(\tau)[]$
struct	$\text{struct}\{\tau_0;\ldots;\tau_n;\}$	$\text{struct}\{\mathbf{st}(\tau_0);\ldots;\mathbf{st}(\tau_n);\}$
union	$\text{union}\{\tau_0;\ldots;\tau_n;\}$	$\text{union}\{\mathbf{st}(\tau_0);\ldots;\mathbf{st}(\tau_n);\}$
pointer	$\tau* : \mathbf{st}(\tau) \neq \varnothing$	$\text{struct}\{\tau*;\mathbf{st}(\tau)*;\}$
	$\tau* : \mathbf{st}(\tau) = \varnothing$	$\text{struct}\{\tau*;\text{void}*\}$
primitive, function, and void types	τ	$\varnothing$

type for a pointer is defined in row 4 of the table, which states that the shadow type of a pointer type $\tau*$ is a structure containing two pointers. The first pointer is a replica pointer of type $\tau*$. The second pointer has type $\mathbf{st}(\tau)*$ and points to a shadow object. If τ's shadow type is null, then a void$*$ is used as a placeholder. 3) The shadow type for all other types is null. The reason that the remaining shadow types are null is that we need to keep additional information only for pointers.

C. Augmented Types

Shadow data structures are a strategy for mapping application pointers stored in memory to replica and shadow pointers. We also need a way to map application pointers to replica and shadow pointers across function boundaries. DPMR accomplishes that by modifying functions and function types to take and return replica and shadow pointers corresponding to pointer arguments and pointer return values. DPMR transforms all types used in a program to augmented types. The definition for augmented types under SDS is given in column 3 of Table II.[2] The primary effect of transforming an application to use augmented types is that function types are transformed as follows. For every pointer passed to a function, a corresponding replica pointer and shadow pointer are added to the function's type. If a pointer of type r is returned by a function, a pointer to the shadow type of r's augmented type is passed to the function. The memory referenced by that pointer is used to store a replica pointer and shadow pointer corresponding to the function's return value.

D. External Code

DPMR is an interprocedural transformation, meaning that it operates on an entire program. A common problem with interprocedural transformations is that the entire program may not be available for transformation. Such a situation arises if an application uses third-party libraries for which source code is unavailable. DPMR handles external code through the use of *external code support libraries*.

[2]The definition presented here assumes that function arguments and return values are scalars, although it can easily be altered for nonscalar arguments and return values.

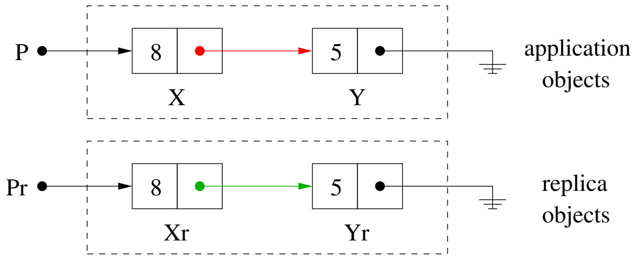


Figure 3: Mirrored Data Structures Example

External code support libraries contain two components. First, they contain wrappers for external functions. A wrapper performs the functionality of an external function as well as DPMR behavior that would occur if the external function had been transformed by DPMR. For example, if an external function stores a value v to a buffer B, the function's wrapper will store v to B's replica. The second responsibility of an external code support library is to allocate space for, and to initialize replica and shadow memory corresponding to, external global variables. DPMR already contains an external code support library for much of `libc`.

III. Mirrored Data Structures

In this section, we present MDS, an alternative to the SDS memory layout. Under MDS, pointers stored in replica memory mirror pointers stored in application memory. Figure 3 illustrates the MDS layout. In application memory, X contains a pointer to Y. Therefore, in replica memory, a pointer to Y's replica is stored in X's replica. While MDS makes the pointers stored to application and replica memory incomparable, it completely eliminates shadow memory.

A. Advantages of MDS

MDS offers three benefits that SDS does not. First, MDS imposes significantly fewer restrictions on input programs than SDS does. In Section III-C, we will see that it eliminates all DPMR restrictions on memory allocation, pointer arithmetic, and pointer-to-pointer casts. The restrictions imposed by MDS are a strict subset of those imposed by SDS. The second advantage of MDS is that it has the potential to induce less overhead than SDS does, because of its reduced memory footprint and reduction in the number of additional loads and stores that are required under its transformations.

Table II: Augmented Type Definition

		SDS	MDS
Description	**Original Type** t	**Augmented Type** $\mathbf{at}(t)$	**Augmented Type** $\mathbf{at}(t)$
array	$\tau[]$	$\mathbf{at}(\tau)[]$	$\mathbf{at}(\tau)[]$
struct	$\text{struct}\{\tau_0;\ldots;\tau_n;\}$	$\text{struct}\{\mathbf{at}(\tau_0);\ldots;\mathbf{at}(\tau_n);\}$	$\text{struct}\{\mathbf{at}(\tau_0);\ldots;\mathbf{at}(\tau_n);\}$
union	$\text{union}\{\tau_0;\ldots;\tau_n;\}$	$\text{union}\{\mathbf{at}(\tau_0);\ldots;\mathbf{at}(\tau_n);\}$	$\text{union}\{\mathbf{at}(\tau_0);\ldots;\mathbf{at}(\tau_n);\}$
pointer	$\tau*$	$\mathbf{at}(\tau)*$	$\mathbf{at}(\tau)*$
function types	$r(\tau_0,\ldots,\tau_n)$	$\mathbf{at}(r)(\mathbf{st}(\mathbf{at}(r))*,$ $\mathbf{at}(\tau_0),\mathbf{rpt}(\tau_0),\mathbf{spt}(\tau_0)),\ldots,$ $\mathbf{at}(\tau_n),\mathbf{rpt}(\tau_n),\mathbf{spt}(\tau_n))$	$\mathbf{at}(r)(\mathbf{rpt}(r)*,$ $\mathbf{at}(\tau_0),\mathbf{rpt}(\tau_0)),\ldots,$ $\mathbf{at}(\tau_n),\mathbf{rpt}(\tau_n))$
primitive and void types	τ	τ	τ
Description	**Original Type** t	**Replica Parameter Type** $\mathbf{rpt}(t)$	**Replica Parameter Type** $\mathbf{rpt}(t)$
pointer	$\tau*$	$\mathbf{at}(\tau)*$	$\mathbf{at}(\tau)*$
other	τ	$\varnothing$	$\varnothing$
Description	**Original Type** t	**Shadow Parameter Type** $\mathbf{spt}(t)$	–
pointer	$\tau* : \mathbf{st}(\tau) \neq \varnothing$	$\mathbf{st}(\mathbf{at}(\tau))*$	
	$\tau* : \mathbf{st}(\tau) = \varnothing$	void$*$	
other	τ	$\varnothing$	

Section V compares MDS and SDS overheads that were computed from experiments. The third advantage of MDS is that the memory footprint for a program under MDS is less than or equal to that of the same program under SDS. They would be equal only if no pointers were stored to memory. In general, the memory overhead for SDS is 2x to 4x, while the overhead for MDS is 2x. Those calculations do not account for the fact that DPMR increases register pressure, which may force more registers to memory. However, the register pressure for SDS is greater than or equal to that of MDS.

B. MDS Transformations

Like SDS, MDS uses augmented types to map application pointers to replica pointers across function boundaries. Under MDS, the definition of augmented types is slightly different, as shown in column 4 of Table II. MDS-augmented types add replica pointers, corresponding to original pointer arguments, to functions. In addition, if a function returns a pointer, MDS-augmented types add an additional pointer parameter, to which a called function can store a replica pointer corresponding to the returned pointer.

Before defining the code transformations that MDS uses, we define the functions $\gamma()$ and $\pi()$.

$$\gamma(r) \equiv \begin{cases} \{r, r_r\} & \text{if } \mathbf{type}(r) = \tau* \\ \{r\} & \text{else} \end{cases}$$

$$\pi(t) \equiv \begin{cases} \{rvRopPtr\} & \text{if } t = \tau* \\ \varnothing & \text{else} \end{cases}$$

$\gamma()$ takes a virtual register and converts it to a set of virtual registers that are used in function calls and declarations under MDS. $\pi()$ takes the type of a function's return value and converts it to a set of arguments that are added to function calls with that return type.

Tables III and IV contain the code transformations that are used by the MDS-based DPMR design. The transformations in Table III add to the original program behavior, while the transformations in Table IV replace original program behavior. Below, we describe each transformation.

- *allocation*: When an allocation instruction occurs in the program under transformation, the program is transformed to allocate an augmented type. In addition, a replica memory buffer is allocated.
- *heap deallocation*: When heap memory is deallocated, MDS inserts an instruction to deallocate the corresponding replica memory.
- *store*: If a non-pointer is stored to memory by the original application, then the same non-pointer value is stored to replica memory. On the other hand, if a pointer is stored to application memory, then the corresponding replica pointer is stored to replica memory. That is a key difference between SDS and MDS.
- *load*: If a non-pointer value is loaded by the original application, then the corresponding value can be loaded from replica memory and compared with the original loaded value. If a pointer is loaded from memory, then the corresponding replica pointer is loaded from replica memory. In the case of a pointer loaded from memory, we do not perform a load comparison, because the pointers are supposed to be different by definition.
- *address of a structure field*: When a program computes the address of a field in a structure, MDS applies identical addressing arithmetic to the corresponding replica pointer.

Table III: MDS Transformation – Added Behavior

Description	Original Behavior		Added Behavior
heap deallocation	`free(`p`)`		`free(`p_r`)`
store	$\star p \leftarrow x$	$\mathbf{type}(x) \neq \tau*$	$\star p_r \leftarrow x$
		$\mathbf{type}(x) = \tau*$	$\star p_r \leftarrow x_r$
load	$x \leftarrow \star p$	$\mathbf{type}(x) \neq \tau*$	`assert(`x `==` $\star p_r$`)`
		$\mathbf{type}(x) = \tau*$	$x_r \leftarrow \star p_r$
address of a struct field	$x \leftarrow$ `&(`p`->`f_i`)`		$x_r \leftarrow$ `&(`p_r`->`f_i`)`
address of an array element	$x \leftarrow$ `&`p`[`i`]`		$x_r \leftarrow$ `&`p_r`[`i`]`
address of a function	$p \leftarrow$ `&`fun		$p_r \leftarrow$ `&`fun

Table IV: MDS Transformation – Transformed Behavior

Description	Original Behavior		Transformed Behavior
allocation	$p \leftarrow allocate\,(\tau)$: $allocate = \{$`malloc`$\|$`alloca`$\|$`global`$\}$		$p \leftarrow allocate\,(\mathbf{at}(\tau))$ $p_r \leftarrow allocate\,(\mathbf{at}(\tau))$
pointer-to-pointer cast	$q \leftarrow (\tau*)\,p$		$q \leftarrow (\mathbf{at}(\tau)*)\,p$ $q_r \leftarrow (\mathbf{at}(\tau)*)\,p_r$
function declaration	$fun\,(a_0, \ldots, a_n)$: $\mathbf{type}(fun) = r(\tau_0, \ldots, \tau_n)$		$fun\,(\pi(r) \cup (\bigcup_{i=0}^{n} \gamma(a_i)))$: $\mathbf{type}(fun) = \mathbf{at}(r(\tau_0, \ldots, \tau_n))$
function call	$x \leftarrow fun\,(p_0, \ldots, p_n)$: $\mathbf{type}(fun) = r(\ldots)$	$r = \tau*$	$rvRopPtr \leftarrow$ `alloca(`$\mathbf{at}(r)$`)` $x \leftarrow fun\,(\{rvRopPtr\} \cup (\bigcup_{i=0}^{n} \gamma(p_i)))$ $x_r \leftarrow \star rvRopPtr$
		$r \neq \tau*$	$x \leftarrow fun\,(\bigcup_{i=0}^{n} \gamma(p_i))$
function return	`return` x	$\mathbf{type}(x) = \tau*$	$\star rvRopPtr \leftarrow x_r$ `return` x
		$\mathbf{type}(x) \neq \tau*$	`return` x

- *address of an array element*: If a program addresses an array element, then similar addressing arithmetic is applied to the corresponding replica pointer.
- *pointer-to-pointer cast*: If a cast instruction from one pointer type to another pointer type occurs, it is transformed into a cast to an augmented pointer type. A similar cast to an augmented pointer type is applied to the corresponding replica pointer.
- *address of a function*: When a pointer is assigned the address of a function, a replica pointer is assigned the same address. Since functions are never explicitly dereferenced, i.e., they are only dereferenced via a function call, there is no corresponding replica function from which to compute an address. Thus, we simply use the same address as the application pointer.
- *function declaration*: Function declarations are transformed to use augmented function types, which potentially add new arguments.
- *function call*: Function calls are transformed so that replica pointers are passed to the called function for every original pointer parameter. In addition, if the called function returns a pointer, then memory must be allocated to hold a replica pointer corresponding to the return value. A pointer to that memory is passed to the called function. When the called function returns, the replica pointer, corresponding to the return value, is loaded from memory.
- *function return*: If a function returns a pointer, then a store instruction must be inserted before each of the function's return instructions. The store writes a replica pointer, which corresponds to the function's return value, to the memory pointed to by the *rvRopPtr* argument.
- *global variable initialization*: Global variable initialization can be thought of as a series of non-pointer operations, pointer arithmetic, and store operations that

can be executed at compile time. Those operations can be transformed using the rules discussed above.

C. Comparison of MDS and SDS Limitations

As discussed above, one of the major benefits of MDS is that it significantly reduces the restrictions imposed by the SDS-based DPMR design. Next, we provide the first precise articulation of the restrictions imposed by SDS and compare them to those imposed by MDS.

- *memory allocation*: At a memory allocation site a, SDS requires that the allocated type t satisfy the equation below, such that T is the set of all types that the result of a could hold during execution.

$$\mathbf{sizeof}(\mathbf{st}(\mathbf{at}(t))) \geq \max_{\tau \in T}(\mathbf{sizeof}(\mathbf{st}(\mathbf{at}(\tau))))$$

  The above requirement ensures that the amount of shadow memory allocated at a will be large enough to support any type to which the allocated memory may be cast during execution. For example, such a restriction would prevent us from allocating an array of 8 characters and later storing a pointer in that allocated memory. Since shadow memory is no longer an issue, *MDS does not impose any restrictions on memory allocations.*
- *loads and stores*: Under SDS, pointers that are loaded from or stored to memory must be typed as pointers when loaded and stored,[3] and non-pointer values that are loaded from or stored to memory must be typed as non-pointers, when loaded and stored. Those requirements ensure that SDS loads from and stores to shadow memory in the prescribed manner. MDS also requires that pointers loaded from or stored to memory be typed as pointers, when loaded and stored; however, it does not require that non-pointers loaded from or stored to memory be typed as non-pointers, when loaded and stored. MDS makes the requirement on pointers so that they will be stored correctly to replica memory.
- *structure and array pointer arithmetic*: SDS imposes a restriction on the type of a pointer p, to which structure or array pointer arithmetic is applied. The restriction ensures that pointer arithmetic maintains distinct shadow data locations for each pointer location in a given application object, guaranteeing that shadow memory can be appropriately addressed. To precisely define that restriction, let $\sigma()$ be a function such that $\sigma(t)$ is a structure that is composed only of scalar types and is structurally equivalent[4] to t. Let $\mathbf{type}(p) = s*$ and $\sigma(\mathbf{at}(s)) = \text{struct}\{t_0\ f_0; \ldots; t_n\ f_n;\}$. Let i be the index of the field to which the pointer arithmetic advances p. Let D be the set of types to which p may point at runtime. For each $d \in D$, where $\sigma(\mathbf{at}(d)) = \text{struct}\{\tau_0\ e_0; \ldots; \tau_m\ e_m;\}$, we require that there exist a field e_j such that

$$\mathbf{sizeof}(\text{struct}\{t_0; \ldots; t_{i-1};\}) = \mathbf{sizeof}(\text{struct}\{\tau_0; \ldots; \tau_{j-1};\}).$$

  If such a field does exist, then we also require that

$$\sum_{k=0}^{i-1} \text{isPointer}(t_k) = \sum_{l=0}^{j-1} \text{isPointer}(\tau_l).$$

  On the other hand, *MDS imposes no restrictions on structure and array pointer arithmetic*. That is one of the biggest advantages of MDS over SDS. Restrictions on pointer arithmetic are not required because any type-generic pointer arithmetic that is applied to application memory can be applied to replica memory. That is the case because the layouts of an application object and its corresponding replica object are structurally identical.
- *pointer-to-pointer casts:* For a pointer-to-pointer cast $p = (\alpha*)q$, where $\mathbf{type}(q) = \beta*$, SDS requires a type restriction if q originated from pointer arithmetic. Specifically, if

$$\sum_{k=0}^{n} \text{isPointer}(t_k) = 0 : \sigma(\mathbf{at}(\beta)) = \text{struct}\{t_0; \ldots; t_n;\},$$

  then we require that

$$\sum_{l=0}^{m} \text{isPointer}(\tau_k) = 0 : \sigma(\mathbf{at}(\alpha)) = \text{struct}\{\tau_0; \ldots; \tau_m;\}.$$

  That restriction helps ensure that a distinct shadow data location exists for each pointer location in a given application object. Like structure and array pointer arithmetic, *MDS imposes no restrictions on pointer-to-pointer casts.*
- *int-to-pointer casts*: Both SDS and MDS forbid int-to-pointer casts. Such casts are not allowed under SDS because DPMR would have no way to set corresponding shadow pointers. They are not allowed under MDS because DPMR would have no way to set corresponding replica pointers.
- *external code*: SDS and MDS require that external code support libraries be implemented for all external code with which an input program interacts. In general, MDS external code support libraries are easy to write. Many times, an external function wrapper simply calls the corresponding external function once with pointers to application memory and once with pointers to replica memory.

One might expect that MDS would reduce the class of faults that DPMR can detect, since it eliminates pointer comparability. However, MDS and SDS detect the same class of faults. One reason that MDS does not restrict

[3]Pointers do not need to be precisely or accurately typed.

[4]In this context, pointers and non-pointers are not structurally equivalent.

DPMR's fault class is that DPMR detects the corruptions that result from memory errors. The type of memory that is corrupted by a memory fault, i.e., a pointer, non-pointer, or both, is arbitrary. Therefore, restricting the comparison space (by eliminating pointer comparisons) does not restrict the space of error sources.

Another reason that MDS does not restrict DPMR's fault class is that corrupted pointers often lead to indirect detection. Consider the memory layout in Figure 1. Let object B be a pointer to an integer X. If a buffer overflow spills out of object A, it will corrupt a pointer to X (object B) in application memory and object C in replica memory. Suppose that the pointer to X, stored in B, is used to load X. The value for X from application memory will likely deviate from the value from replica memory, because the application pointer to X is corrupted, while the replica pointer is not corrupted. MDS will recognize that the values are different and report an error.

Although some limitations remain under MDS, those limitations are only an issue for applications that treat pointers and integers as interchangeable. However, even those limitations can be overcome with the static memory analyses presented in [9]. One of the primary goals of those analyses is to eliminate all DPMR integer comparisons whose sources could be directly derived from pointers. Doing so eliminates the false positives that would arise if pointers (masquerading as integers) were compared under MDS.

IV. Experimental Framework

In this section, we describe the framework that was used to experimentally evaluate the MDS and SDS approaches. Our goal was to compare MDS to SDS with respect to dependability and performance. Dependability results were computed from runs of an application with and without DPMR, in the presence of fault injections. We computed performance results by comparing fault-free runs of an application when it had not been transformed by DPMR to fault-free runs when it had been transformed by DPMR. Below we discuss how the fault injection experiments were conducted, what measures were computed from the experiments, the applications that were evaluated, and the environment under which those experiments were conducted.

A. Fault Injection Experiments

As mentioned above, fault injection experiments were used to evaluate the dependability properties of DPMR variants. Since the focus of DPMR is on the detection of software errors, our fault injection framework was designed to simulate software bugs. Many fault injection strategies inject a fault once during the runtime of an experiment. However, software bugs cause a fault to be executed every time a faulty piece of code executes. Therefore, we injected faults by directly modifying the original application code. Under our strategy, faulty code executed every time an injection site executed.

We injected two types of faults. The first were *heap array resizes*, which reduce the number of objects requested at a heap array request. By reducing the amount of memory requested by a heap array request, we were injecting a fault that could lead to a buffer overflow. The heap array resizes reported in this paper reduced a given heap allocation by 50%. The second type of faults injected were *immediate frees*. Immediate frees deallocate heap memory as soon as it is allocated. Immediate frees lead to dangling pointer errors such as reads, writes, and frees after free. In each fault injection experiment, we injected exactly one fault at exactly one injection location. For heap array resizes, any heap array request was a candidate injection site, while all heap allocation sites were candidates for immediate free injections. Dependability measures are computed from experiments that uniformly and deterministically injected faults over the space composed of each pairing of a fault injection type to a valid injection location.

B. Measures

Throughout our experiments, we evaluated MDS and SDS under various diversity transformations and state comparison policies. Diversity transformations were applied to replica memory with the goal of causing errors to manifest differently in replica memory and in application memory. State comparison policies determined when loads of comparable memory should be replicated and compared. Specific diversity transformations and state comparison policies are discussed in the following section.

Each experiment constituted one run of one application variant and was defined by the tuple (W, C, D, I, RN). W is the workload under which the application was run. If the application was transformed with DPMR, C is the state comparison policy that was applied to the application, and D is the diversity transformation that was applied to the application's replica memory. If the application was not transformed by DPMR, C and D are null. I is the fault that was injected during the experiment. For non-fault injection experiments, I is null. Finally, RN is the run index under the other configuration parameters.

Table V specifies the notation used in measure definitions. Three concepts from the table require further explanation. First, a *successful fault injection* occurs when the injected faulty code executes at least once during the execution of the experiment. A successful fault injection does not imply that an error has manifested. In fact, we cannot accurately determine when an error has manifested without intrusive instrumentation. Second, *correct output* indicates that a run of an application variant produced the same output that an untransformed, fault-free version of the application would have produced. Incorrect output includes "bad" results as well as error detection and experiment timeouts. Finally,

Table V: Measurement Components

Notation	
$\mu[X]$	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

natural detection is error detection that is native to the application under study. Natural detection is defined on an application-by-application basis and may include program crashes, assertion violations, and error messages.

Using the components discussed above, we define the experiment measures that we use in this paper.

- *coverage*: In successful fault injection experiments, a fault is covered if it results in correct output, natural detection, or DPMR detection. Thus, coverage equals
$$\mu[CO \vee Ndet \vee Ddet | C = c, D = d, F, SF].$$
- *overhead*: We define overhead in fault-free experiments as the ratio of a variant's running time to the running time of the application without DPMR. Formally, overhead is
$$\frac{\mu[T|C = c, D = d, \neg F]}{\mu[T|C = \varnothing, D = \varnothing, \neg F]}.$$

C. Remaining Details

All experiments were conducted using a DPMR compiler built on the LLVM compiler framework [10]. Four benchmarks from the SPEC CPU2000 benchmark suite[5] [11] were utilized, namely art, bzip2, equake, and mcf. Each experiment was run on a machine with a 2 GHz AMD Athlon processor, 512 MB of memory, and a 256 KB L2 cache. Each machine was unloaded when experiments were conducted, and timing measures refer to wall clock time. Timeouts were set for each experiment at approximately 20 times the normal application running time, under the specified workload. If a timeout occurred, the experiment is considered to have produced undetected incorrect output.

V. Experimental Results

In this section, we describe the diversity transformations and state comparison policies that were used in our experimental evaluations. We follow their description with a comparison of experimental results under MDS and SDS.

[5]SPEC CPU2000 was chosen over SPEC CPU2006 so that results in this paper would be directly comparable with results from [8].

A. Diversity Transformations

MDS and SDS were evaluated under several diversity transformations. The transformations explicitly introduced diversity into replica behavior. The first diversity transformation was *pad-malloc*. Pad-malloc increases the request size for all replica heap allocation requests by a static amount. Pad-malloc was chosen explicitly to target buffer overflows. Buffer overflows out of application objects can immediately corrupt other objects, whereas the initial portion of an overflow out of a replica will write to unused padding.

The second diversity transformation that DPMR utilized in the experiments was *zero-before-free*. Zero-before-free writes zeros to the bytes of a replica buffer immediately prior to deallocation. Zero-before-free was chosen because of its potential to cause dangling pointer errors to manifest differently. In particular, if a read occurs to a deallocated object X before X and its replica X_r are reallocated, the read from X will return data, while the read from X_r will return a zero value.

The final diversity transformation used was *rearrange-heap*. Under rearrange-heap, when an application object X is allocated on the heap, its replica is given the location that it would receive if a random number of copies of X were allocated just prior to the replica's allocation. The random number is chosen uniformly between 1 and 20. Rearrange-heap is designed to detect dangling pointer errors. By randomizing the location of replica objects, it decreases the likelihood that an application object X and its replica X_r will occupy the same pair of memory locations as a previously allocated (and freed) application object Y and its replica Y_r. If X is given the memory that Y previously occupied and a dangling pointer to Y and Y_r exists, then when that dangling pointer is used, the result will be a function of X but will most likely not be a function of X_r.

DPMR's intra-process partial replication scheme produces implicit diversity as well as explicit diversity. For example, if an application allocates an object X followed by an object Y, the objects and their replicas may be consecutively allocated in the following order: X, X_r, Y, and Y_r. Under such a layout, the object following X is not paired with the object following X_r. Therefore, if X suffers from a buffer overflow, the overflow out of X will corrupt X_r, and the overflow out of X_r will corrupt Y. Such an error could be detected by comparing Y and Y_r. When we rely solely on implicit diversity, we say that we have *no-diversity*.

B. State Comparison Policies

We now examine three state comparison policies that DPMR deploys. The first is the *all loads* policy, in which all application loads that are comparable are replicated and compared. Under SDS, all loads are comparable. Under MDS, only non-pointer loads are comparable, since the pointers stored to application and replica memory are different. The second comparison policy is *static load-checking*.

Static load-checking replicates and compares a spatial fraction of comparable loads. The final comparison policy is *temporal load-checking*. Under temporal load-checking, a temporal fraction of comparable loads are replicated and compared at runtime.

When evaluating comparison policy results, it is important to keep in mind that the pools of loads from which the temporal load-checking and static load-checking policies are chosen are different for MDS and SDS. Let P be the number of loads of pointers in a program, and let N be the number of non-pointer loads. If we only include a fraction β of the load checks, then under SDS we will have removed $(1-\beta)(P+N)$ load checks. However, under MDS we will have removed only $(1-\beta)N$ load checks. In terms of reduction of overhead, we therefore expect that a switch from the SDS all loads policy to an SDS policy that checks a β fraction of comparable loads will result in greater improvement than could be obtained by switching from the MDS all loads policy to an MDS policy that checks a β fraction of comparable loads.

C. Dependability Results

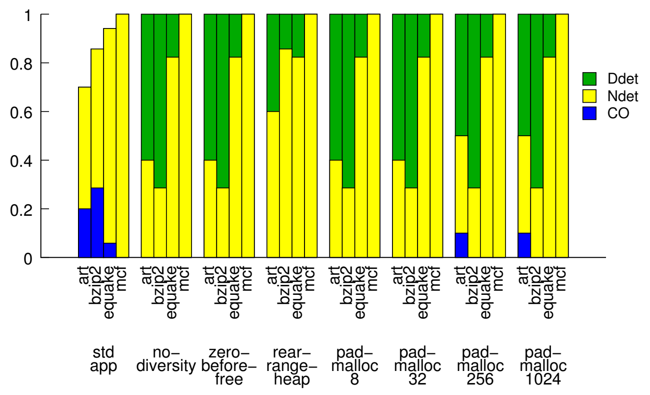
We turn our attention to experimental results, concentrating on dependability first, since the benefits of MDS will lose value if MDS is not able to achieve reasonable dependability. We begin by examining diversity transformations under the all loads comparison policy, in which all comparable application loads are replicated and compared. Figure 4 displays diversity transformation coverage results under heap array resizes and immediate frees. The results in the figure are similar to the SDS coverage results presented in [8].

In particular, we see that implicit diversity (indicated in the figures as no diversity) was enough to cover the buffer overflows caused by heap array resizes. As previously discussed, implicit diversity makes it unlikely that the object following an object X will be paired with the object following X's replica. Therefore, a buffer overflow of X will likely corrupt unpaired objects, making the corruption detectable.

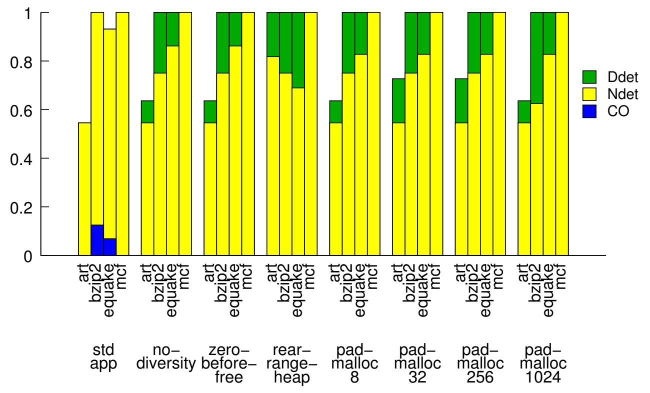
The second result that we obtain from the diversity transformations is that the rearrange-heap mechanism is the only diversity mechanism to cover all injected faults. Rearrange-heap is able to detect buffer overflows for the same reasons that implicit diversity can. Rearrange-heap is able to detect dangling pointers because it makes it unlikely that an object X and its replica will occupy the same memory as a previously allocated and freed object-replica pair.

Figure 5 shows coverage results for different state comparison policies. The state comparison policies are evaluated under the rearrange-heap diversity transformation. Again, the results are similar to those for SDS, found in [8]. Coverage remains fairly robust in the face of reduced checking. In fact, for temporal load-checking, coverage never dipped when

[6]Refer to Table V for definitions of *Ddet*, *Ndet*, and *CO*.



(a) Heap Array Resizes



(b) Immediate Frees

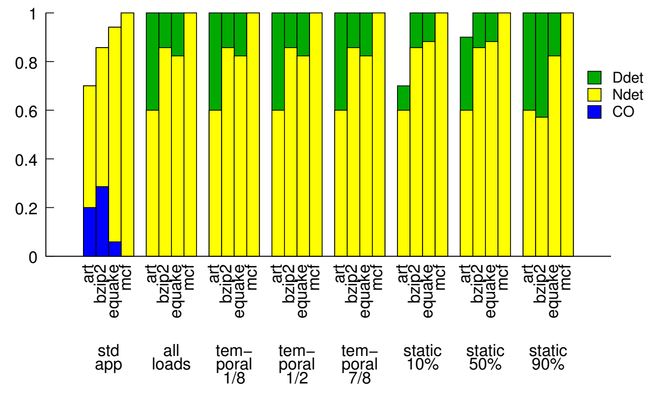
Figure 4: Coverage of MDS Diversity Transformations Under the All Loads Comparison Policy[6]

we reduced the checking frequency. When the checking frequency was reduced for static load-checking, we observed some drops in coverage. Unfortunately, that is the nature of static load-checking. A reduction of 50% of the load checks at compile time could end up removing nearly all or almost none of the load checks during runtime. The drop in coverage of the 10% and 50% static load-checking policies could also be an indication that certain faults end up impacting only certain load sites.

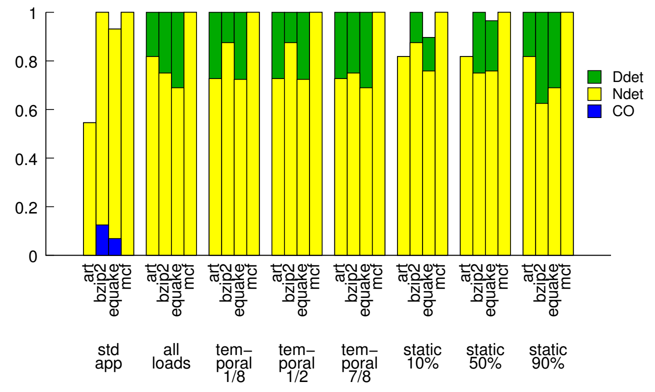
As noted above, the coverage results for MDS are very similar to those for SDS, leading us to conclude that the benefits of MDS are not diluted by coverage losses.

D. Performance Results

We now evaluate the ability of MDS to reduce DPMR's overhead when compared to SDS. Figure 6 displays (execution time) overhead results for the different diversity transformations. Each diversity transformation is evaluated using the all loads comparison policy. In all cases except one (art pad-malloc 32), MDS outperformed its SDS counterpart, albeit marginally so for art and bzip2. Examination of the code for the four benchmarks indicates that the fractions of



(a) Heap Array Resizes



(b) Immediate Frees

Figure 5: Coverage of MDS State Comparison Policies Under Rearrange-Heap Diversity[6]

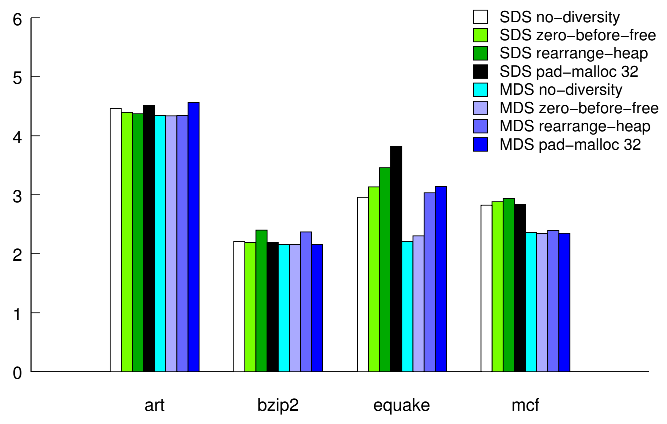


Figure 6: Overhead Comparison of Diversity Transformations Under the All Loads Comparison Policy

the allocations that are for memory to hold pointers are larger for equake and mcf than for art and bzip2. That observation fits the overhead results. If very few pointers are stored to memory, then the executions of SDS and MDS are very similar. Without pointers in memory, SDS does not need to

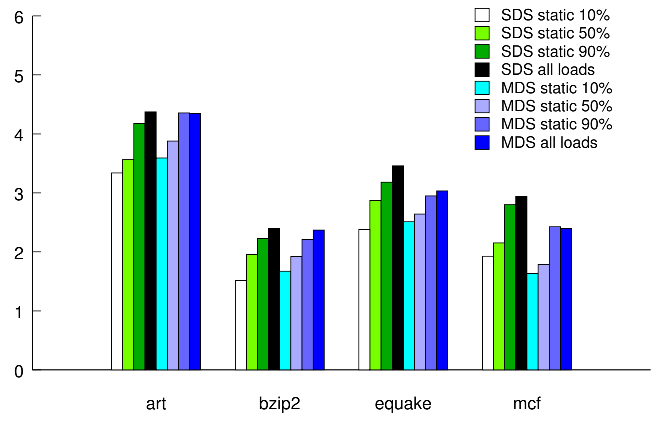


Figure 7: Overhead Comparison of Static Load-Checking Policies Under Rearrange-Heap Diversity

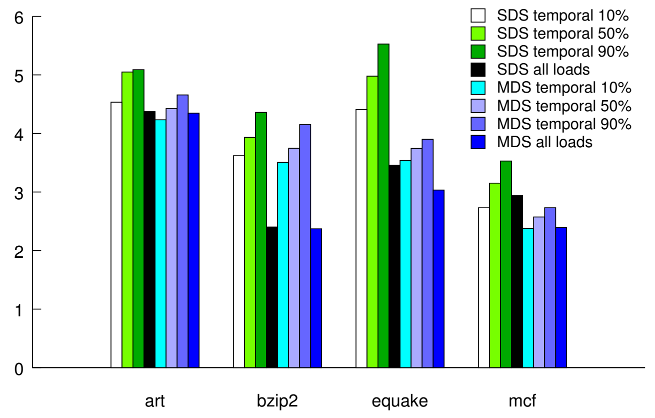


Figure 8: Overhead Comparison of Temporal Load-Checking Policies Under Rearrange-Heap Diversity

allocate shadow memory, and everything in memory will be comparable under MDS. Therefore, the overheads of both memory layouts will be approximately the same. For the more pointer-heavy benchmarks, equake and mcf, MDS is able to achieve stronger overhead gains. The MDS diversity transformation overheads for equake are 74% to 89% of their SDS counterparts, and the MDS mcf overheads are 81% to 84% of the SDS overheads.

We now compare overhead results for state comparison policies. State comparison policies are evaluated using the rearrange-heap diversity transformation. Those results can be seen in Figures 7 and 8. For the static load-checking policies, the pointer-heavy benchmarks, equake and mcf, tend to see performance gains from MDS. For bzip2, the overheads under MDS tend to be about the same as for SDS. For art, SDS tends to outperform MDS; however, we attribute that to caching effects and the random nature of the load-checking selection process under static load-checking.

For the temporal load-checking policies, we see that the MDS versions always outperformed their SDS counterparts.

However, the temporal load-checking policies generally performed worse than the all loads policy. As noted in [8], the reason that the reduced checking of temporal load-checking does not result in performance improvements is that it imposes additional runtime overheads in deciding when to make checks. We feel that the periodic nature of temporal load-checking could be exploited in future work to produce temporal load-checking policies that achieve performance improvements.

VI. Related Work

Software memory errors have long plagued programmers, and much dependability research has been focused on them. Techniques that address both spatial and temporal memory errors can be divided into five categories: runtime checking, static analysis and runtime checking, randomized memory layouts, error correction, and diverse replication.

Runtime checking: Runtime checking methods maintain metadata and use them to detect memory errors by performing checks at runtime. Such methods include Valgrind's Memcheck [12], Purify [13], Safe-C [14] and its variants [15] and [16], and BGI [17]. Unfortunately, Memcheck and Purify suffer from large overheads, and neither can detect dangling pointers to reallocated memory or out-of-bounds pointers to valid memory, both of which are error classes that DPMR can detect. Safe-C uses fat pointers that are not compatible with library code, and it is not capable of detecting uninitialized reads. Patil and Fischer [15] and Xu et al. [16] do not use fat pointers, but neither approach allows arbitrary pointer-to-pointer casts, a restriction overcome by MDS. BGI is a slightly different approach that utilizes access control lists but is not intended to enforce internal memory properties, such as the absence of array bounds violations and dangling pointer errors.

Static analysis and runtime checking: Some techniques utilize runtime checking but try to minimize its use with static analysis. Two such tools are CCured [18] and SAFECode ([19], [20], and [21]). CCured utilizes fat pointers (which, as mentioned before, are incompatible with library code) and does not detect uninitialized reads. On the other hand, SAFECode, in order to handle both spatial and temporal memory errors, imposes significant overhead during memory allocation and deallocation and utilizes a memory pool strategy that may exhaust virtual memory.

Randomized memory layouts: Randomized memory layouts are used by DieHard [22] and its variants [23] and [24] as well as by a similar technique called Heap Server [25]. DieHard and Heap Server utilize randomized memory layouts to avoid memory errors (which is different from DPMR's goal of detection) and do not protect the stack or global variables. In order for DieHard and similar techniques to handle uninitialized reads, they require replication. When replication is utilized, DieHard and DPMR can complement each other. For example, DieHard's memory randomization could be utilized by DPMR, and DPMR's partial replication strategy could be utilized by DieHard.

Error correction: Two projects have aimed at error correction. The first is Exterminator [26]. Programs using Exterminator dump heap images when memory errors are detected by the use of fence posts and canary values. When several heap images have been collected, Exterminator statistically analyzes them to create program patches to avoid errors in the future. The second project that aims to correct errors is Rx [27]. Rx is a checkpoint-rollback scheme that, upon detecting an error, rolls a program back to a safe checkpoint. It then re-executes the program in a diverse environment that is designed to avoid the error in the second execution. Error correction is complementary to DPMR. DPMR's strong detection capabilities make it an excellent complement to both Exterminator and Rx.

Diverse replication: Finally, diverse replication is the category under which DPMR falls. Another approach in the category is Samurai [28]. Under Samurai, a programmer identifies regions of heap memory that are critical. Those memory regions are replicated, and DieHard is used to diversify them. One difference between Samurai and DPMR is that Samurai requires programmer support, while DPMR is automated. A second difference is that DPMR covers memory error classes that Samurai does not, such as uninitialized reads and double frees.

VII. Conclusion

In our previous work [8], we proposed an approach, called Diverse Partial Memory Replication (DPMR), for detecting both spatial and temporal software memory errors in all segments of an application's data memory. One of the key technical challenges for DPMR is that of handling pointers stored in memory. In order to maximize coverage, we developed a memory layout, called Shadow Data Structures (SDS), that makes pointers stored in memory comparable. After discovering that SDS achieves extremely high coverage levels, we proposed in this paper an alternative design called Mirrored Data Structures (MDS). MDS potentially sacrifices coverage, by making pointers incomparable, in order to achieve the following gains. 1) MDS greatly reduces the restrictions that DPMR imposes on input programs. In particular, it eliminates all typing constraints on memory allocation, pointer arithmetic, and pointer-to-pointer casts. 2) MDS has the potential to reduce DPMR's overhead. In experimental results, we saw that for the pointer-heavy benchmarks that we evaluated, MDS reduced DPMR overhead by 11% to 26%. 3) MDS significantly reduces DPMR's memory footprint. Although one might think that MDS reduces the class of faults that DPMR can detect, since it makes pointers incomparable, MDS and SDS can detect the same class of faults.

To this point, the focus of our research has been on evaluating the effectiveness of the DPMR approach. We believe that the performance of both the SDS and MDS approaches can be optimized with aggressive compiler optimizations. For example, we could hide some of the latency associated with fetching replica and shadow memory by forcing associated application, replica, and shadow objects to fall on the same memory page. Additionally, the latency of fetching replica memory and shadow memory could be hidden through a combination of memory prefetching and memory comparison delay.

Acknowledgments

The authors thank Jenny Applequist and Eric Rozier 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] NIST: National Institute of Standards and Technology. (2002, Jun.) Software errors cost U.S. economy $59.5 billion annually: NIST assesses technical needs of industry to improve software-testing. Note: accessed Apr. 2010. [Online]. Available: http://www.nist.gov/public_affairs/releases/n02-10.htm

[2] R. N. Charette, "Why software fails," *IEEE Spectrum*, vol. 42, no. 9, pp. 42–49, Sep. 2005.

[3] J. Nelißen. (2002, May) Buffer overflows for dummies. SANS Institute. [Online]. Available: http://www.sans.org/reading_room/whitepapers/threats/buffer-overflows-dummies_481

[4] CERT. (2003, Mar.) CERT advisory CA-2003-02 double-free bug in CVS server. [Online]. Available: http://www.cert.org/advisories/CA-2003-02.html

[5] ——. (2004, Sep.) Vulnerability note VU#350792 MIT Kerberos krb524d insecurely deallocates memory (double-free). [Online]. Available: http://www.kb.cert.org/vuls/id/350792

[6] iDefense Labs. (2007, Jul.) Opera software Opera web browser BitTorrent dangling pointer vulnerability. [Online]. Available: http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=564

[7] J. Afek and A. Sharabani. (2007, Aug.) Dangling pointer: Smashing the pointer for fun and profit. Watchfire. [Online]. Available: http://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf

[8] R. M. Lefever, V. S. Adve, and W. H. Sanders, "Diverse partial memory replication," in *Proc. of the 40th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, Jun. 2010, pp. 71–80.

[9] R. M. Lefever, "Diverse partial memory replication," Ph.D. dissertation, University of Illinois at Urbana-Champaign, April 2011.

[10] LLVM Project. (2012, Oct.) The LLVM compiler infrastructure. [Online]. Available: http://www.llvm.org/

[11] Standard Performance Evaluation Corporation. (2007, Jun.) SPEC CPU2000. [Online]. Available: http://www.spec.org/cpu2000/

[12] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proc. of the USENIX 2005 Technical Conf.*, Apr. 2005, pp. 17–30.

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

[14] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," *ACM SIGPLAN Notices*, vol. 29, no. 6, pp. 290–301, Jun. 1994.

[15] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in C programs," *Software: Practice and Experience*, vol. 27, no. 1, pp. 87–110, Jan. 1997.

[16] 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*, Nov. 2004, pp. 117–126.

[17] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proc. of the 22nd ACM Symp. on Operating System Principles*, Oct. 2009, pp. 45–58.

[18] 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 Systems*, vol. 27, no. 3, pp. 477–526, May 2005.

[19] 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*, Jun. 2006, pp. 144–157.

[20] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for C with very low overhead," in *Proc. of the 28th Int. Conf. on Software Engineering*, May 2006, pp. 162–171.

[21] ——, "Efficiently detecting all dangling pointer uses in production servers," in *Proc. of the 2006 Int. Conf. on Dependable Systems and Networks*, Jun. 2006, pp. 269–280.

[22] E. Berger and B. Zorn, "Diehard: Probabilistic memory safety for unsafe languages," in *Proc. of the ACM SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, Jun. 2006, pp. 158–168.

[23] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn, "Archipelago: Trading address space for reliability and security," in *Proc. of the 13th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008, pp. 115–124.

[24] G. Novark and E. D. Berger, "DieHarder: Securing the heap," in *Proc. of the 17th ACM Conf. on Computer and Communications Security*, Oct. 2010, pp. 573–584.

[25] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, and M. Prvulovic, "Comprehensively and efficiently protecting the heap," in *Proc. of the 12th Int. Symp. on Architecture Support for Programming Languages and Operating Systems*, Oct. 2006, pp. 207–218.

[26] 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*, Jun. 2007, pp. 1–11.

[27] 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 Systems Principles*, Oct. 2005, pp. 235–248.

[28] K. Pattabiraman, V. Grover, and B. G. Zorn, "Samurai: Protecting critical data in unsafe languages," in *Proc. of the 3rd ACM European Conf. on Computer Syststems*, Apr. 2008, pp. 219–232.