An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs[*]

Wei Xu, Daniel C. DuVarney, and R. Sekar
Department of Computer Science, Stony Brook University
Stony Brook, NY 11794-4400
{weixu,dand,sekar}@cs.sunysb.edu

ABSTRACT

Memory-related errors, such as buffer overflows and dangling pointers, remain one of the principal reasons for failures of C programs. As a result, a number of recent research efforts have focused on the problem of dynamic detection of memory errors in C programs. However, existing approaches suffer from one or more of the following problems: inability to detect all memory errors (e.g., Purify), requiring non-trivial modifications to existing C programs (e.g., Cyclone), changing the memory management model of C to use garbage collection (e.g., CCured), and excessive performance overheads. In this paper, we present a new approach that addresses these problems. Our approach operates via source code transformation and combines efficient data-structures with simple, localized optimizations to obtain good performance.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering;
D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Reliability, Security, Languages

Keywords

Memory Safety, C, Program Transformation

1. INTRODUCTION

The C programming language is commonly used for systems programming because of its speed and the precise control it provides over memory allocation. Unfortunately, this control is more than most programmers can fully manage, as evidenced by the profligate frequency of bugs such as memory leaks, dangling pointers, and buffer overruns in commercial C programs. Such memory errors are one of the most common reasons for software failures. Moreover, these errors are hard to track down, and hence contribute significantly to the effort needed for testing and debugging C programs. Even worse, memory errors are the culprit behind most of today's security vulnerabilities in software. Consequently, a number of research efforts have focused on the problem of detection/elimination of this class of errors.

[*]This research is supported in part byNSF grants CCR-0098154 and CCR-0208877, and an ONR grant N000140110967.

Intuitively, a memory error occurs in C programs when the object accessed via a pointer expression is different from the one intended by the programmer. The intended object is called the *referent* of the pointer. Memory errors can be broadly classified into *spatial errors* and *temporal errors*, as described below.

A spatial error occurs when dereferencing a pointer that is outside the bounds of its referent. Common spatial errors include array bounds errors, dereferencing of uninitialized or NULL pointers, and dereferencing pointers obtained by invalid pointer arithmetic.

A temporal error occurs when dereferencing a pointer whose referent no longer exists (i.e., it has been freed previously). Of particular significance are temporal errors involving a pointer whose referent has been freed and subsequently reallocated to a different, unrelated object. In this case, an access of this pointer can change the unrelated object. Moreover, the change will go undetected until the other object is accessed, which may happen much later in the program's lifetime. This means that the symptom of the error will be spatially and temporally removed from the underlying erroneous pointer operation, making debugging very difficult.

The distinction between spatial and temporal errors is important because much prior work [22, 33] has addressed spatial errors, but only provided limited support for the detection of temporal errors. An easy way to avoid temporal errors is to ignore `free` operations, and rely on garbage collection to free unused memory. In addition, local variables accessed via pointers may need to be moved from the stack to the heap to ensure that they are not freed. This is the approach taken in many recent works such as CCured [26] and Cyclone [19], and has played a significant role in reducing their overheads to an acceptable level. In contrast, approaches that handle temporal errors, such as [20, 2, 17] have suffered high performance penalties, typically ranging from a few hundred percent to a thousand percent. Even among these approaches, it is common to handle just the first class of temporal errors mentioned above, but not the second class (i.e., dangling pointer to reallocated memory). Techniques such as [20] fall in this category.

Based on the above discussion, the problem of developing an *efficient approach* for detecting *temporal* as well as *spatial* memory errors in C programs has remained open. We address this problem in this paper, and present a solution based on source-to-source transformation.

1.1 Benefits of the Approach

The approach described in this paper has the following benefits:

- It can handle most C programs, with almost no source code changes. For instance, our current prototype successfully transformed eighteen benchmark programs ranging in size from a few hundred to 29,000 lines of code. The source code changes needed were due to the use of customized memory management functions, which are not supported in our prototype.
- All spatial and temporal errors are detected.

- No changes are made to the memory allocation model. Freed memory is immediately returned to the heap for reuse.
- As compared to previous techniques that handle temporal errors, our approach is faster by at least a factor of two.

The approach described in this paper handles all C programs that do not perform the following operations: (a) make use of customized memory management functions, (b) cast integers to pointers, or (c) cast a pointer to a structure of one type into a pointer to structure of an unrelated type. By (c) we mean that the casting of structures must follow an upcast/downcast paradigm. An upcast of a structure is a cast to another structure which is shorter in length, while downcast denotes the inverse operation. An upcast (or downcast) is permitted by our transformation, provided the larger structure contains pointer fields at all and only the offsets where pointers are present in the shorter structure. C programs typically do not violate these restrictions, as shown by the fact that all the benchmark programs described in Section 5 were successfully handled while requiring almost no source code modifications. Even so, it is possible to relax these restrictions, as discussed in Section 7.

A key feature of our approach is that for programs which obey the casting restrictions, all spatial and temporal errors are promptly detected, without changing the memory allocation model. This has some important benefits over the approaches used by CCured [26] and Cyclone [19], which rely on a garbage collector (specifically, the Boehm-Demers-Weiser conservative garbage collector [6]) to ensure temporal memory safety. Garbage collection makes the runtime overhead of a program unpredictable, negating one of the primary benefits of C, namely, the fact that the performance characteristics of a C program are directly manifest in the source code. Moreover, the nature of the C language requires the garbage collector to use a conservative approach, which means that some memory may not be reclaimed, resulting in memory leaks. These factors make it impractical to switch to a garbage collection model for several classes of applications, such as OS kernel or applications with strict response-time constraints. For these classes of applications, the approach presented in this paper will be more appropriate.

1.2 Organization of the Paper

Section 2 provides an overview of our basic transformation. An extension of this technique to handle casts is presented in Section 3. Simple, local optimizations to reduce performance overheads are discussed in Section 4, followed by experimental results in Section 5. Related work is discussed in Section 6. Section 7 discusses extensions to the approach, and summarizes the results of the paper.

2. BASIC TRANSFORMATION

Our basic transformation is conceptually similar to that of Patil and Fischer [28], which in turn is closely related to the Safe-C approach [2]. However, the specific details in the description below are based on our implementation.

The Safe-C approach maintains extra information (henceforth called metadata) with each pointer variable to detect spatial and temporal errors at runtime. Safe-C uses *fat pointers*, which store metadata together with the pointer. Unfortunately, this changes the space requirements for a pointer, thereby changing structure layouts. This creates a number of compatibility problems:

- *Breaks many programs.* Systems programs often make assumptions regarding the size of pointers and the layouts of structures. For instance, they may assume that the size of integers and pointers will be the same. Another typical assumption is that, in a union consisting of an integer and pointer, a value stored as a pointer may be read as an integer. These assumptions are no longer valid when normal pointers are replaced by fat pointers, and hence such programs need to be modified before they will work correctly with the transformation.
- *Compatibility with libraries.* Since fat pointers change structure layouts, compatibility with precompiled libraries is lost. (Such libraries will access fields in structures using offsets based on untransformed layout of structs, and hence will access incorrect fields or maybe the metadata stored within the fat pointers.)

For these reasons, we separate metadata from the pointer. Our approach for maintaining this metadata is described below.

2.1 Metadata Representation and Maintenance

Our technique detects memory access errors at the level of *memory blocks*, which correspond to the least units of memory allocation, such as a global or local variable, or memory returned by a single invocation of `malloc`. It flags memory errors only at a point where a pointer is dereferenced. Other operations, e.g., the assignment of a pointer from an arbitrary expression, won't cause errors.

For a pointer `p`, the following metadata fields associated are maintained in a variable named `p_info`.

- `base`: the base address of the memory block pointed by `p`.
- `size`: the size (in bytes) of the memory block.
- `cap_ptr`, `cap_index`: used to detect temporal errors.
- `link`: a pointer to a structure that contains metadata for pointers that are stored within `*p`. (See Section 2.3 for further information on this field.)

A spatial error is reported when dereferencing a pointer if its value falls outside the interval $[\texttt{base}, \texttt{base} + \texttt{size})$. To detect temporal errors, we first associate a unique capability with each memory block. A capability is created when a block is allocated (on stack or heap), and stored in a *global capability store (GCS)*. When a pointer to a block B is assigned to a variable `p`, a pointer to B's capability is stored in `p_info.cap_ptr`. When the memory block is freed, the associated capability (in the GCS) is marked invalid. Thus, to check for temporal errors, we simply need to check if `cap_ptr` points to a valid capability.

The above approach has the benefit that each capability requires just one bit of storage, but suffers from the drawback that this storage cannot be reused. As a result, a program performing a large number of memory allocations will eventually run out of memory. To support reuse of capabilities, the above technique is modified as follows. Each capability is given a non-zero 32-bit integer index value. (A value of zero denotes an invalid capability.) This value is also stored in a second metadata field `cap_index`. A capability is considered valid for a pointer `p` if `(p_info.cap_index == p_info.cap_ptr->index)`. Now, a capability slot in the GCS can be reused as long as the capability value is changed before each reuse. With a 32-bit capability index, this approach allows capabilities to be reused 2^{32} times before they become unusable. Alternatively, we may allow the index to wrap around (skipping over zero), treating the probability of temporal errors being missed due to such wrap-around as negligible.

2.2 Capability Store Management

Note that global variables are always temporally valid, so they all share a single capability that is stored in a global variable. For heap and stack allocated objects, their capabilities are held in the heap and stack capability stores (HCS and SCS) respectively. The GCS consists of all the above three components.

HCS is an expandable array. Each slot in the array may store a valid capability, or it may be free. Unused slots in the HCS are organized into a linked list called the *free list*. This means that a used HCS slot contains a capability index, whereas an unused slot contains a pointer to next available slot in the HCS free list. To distinguish between the two types of information, we ensure that the least significant bit (LSB) of valid capability indices is always one, whereas the LSB of pointers in the HCS free list is always zero. (This is because the pointers are aligned on a 32-bit boundary.) This means that there are 2^{31} possible capability values.

When a memory block is allocated on the heap, a capability for this block is allocated from the head of the HCS free list. When a memory block is freed, its capability is inserted at the beginning of the HCS free list. Since valid capability index values are different from pointers in the free list, the insertion of a capability into the free list immediately causes it to become invalid.

The SCS is allocated separately from HCS since capabilities on the SCS are allocated in a LIFO manner. Note that all of the local variables for a function are allocated and freed together. Hence, we can use a single capability for all of the local variables (as opposed to one per variable). This *stack frame capability* is pushed on the SCS on a function entry, and popped off on exit.

2.3 Source Code Transformation

2.3.1 Declarations

For each type and pointer variable declared in the original program, additional declarations are introduced in the transformed program for holding metadata, as illustrated below.

Original	Additional
`struct stu {` `int  id;` `char *name;` `};` `struct stu s;`	`struct ptr_info {` `void *base;` `unsigned long size;` `capability *cap_ptr;` `unsigned long cap_index;` `struct ptr_info *link;` `};` `struct stu_info {` `struct ptr_info name_info;` `};` `struct stu_info s_info;`
`struct stu *p;`	`struct {` `void *base;` `unsigned long size;` `capability *cap_ptr;` `unsigned long cap_index;` `struct stu_info *link;` `} p_info;`

For each structure s which contains pointer fields, a corresponding structure s_info is introduced to hold metadata pertaining to the pointer fields of s. For array variables, a corresponding array is introduced to hold metadata pertaining to the elements of the array. For each pointer p, the corresponding metadata variable p_info is initialized so as to ensure that any access to p (before its first assignment) will raise spatial and temporal errors.

2.3.2 Pointer assignments

Pointer assignments fall into three categories: (a) assignment of one pointer value to another, (b) creation of a pointer value through pointer arithmetic, (c) creation of a pointer value through the & operator or malloc. Transformation for pointer assignments is very simple: when q is assigned to p, an assignment p_info = q_info is introduced in the transformed program. For pointer arithmetics such as p++, the link field of p_info is also increased accordingly (more precisely, p_info.link++) in the transformation. Just like invalid pointers are permitted in C programs as long as they are not dereferenced, the link fields are permitted to contain invalid addresses, but are checked before accesses.

For pointer creation, the following transformation is used for a statement that assigns a malloc-returned block to a pointer p. The size and base fields of p_info are initialized from the argument and return values of malloc. In addition, a new capability is allocated for this block on the HCS. A pointer to this capability is stored in p_info.cap_ptr, and its index stored in p_info.cap_index. In addition, the malloc call is modified to allocate additional storage to hold metadata pertaining to pointers that are to be stored within the block. The type of p indicates what fields within the block will contain pointers. This metadata is initialized to indicate that none of the pointers within the malloc'd block are valid.

Note that malloc may be used to allocate storage for an array rather than a single object. The size argument of malloc is used to determine whether the allocation is for a single object or array. In the latter case, p_info refers to an array rather than a single struct. Code must be generated to initialize all of the elements of p_info. In addition, the metadata pertaining to pointers within each of the array elements needs to be initialized.

When pointers are created using the & operator, the situation is very similar. In particular, consider the assignment p = &q. The base and size fields are initialized from the location and size information pertaining to q, which is available from the symbol table. If q is a global variable, then the capability fields of p_info are initialized from the capability shared by all global variables. If q is local, then the capability fields are initialized to a pointer to the top of SCS (i.e. the current frame's capability). If q contains embedded pointers, then a variable q_info would already have been introduced by the transformation. In this case, the link field is initialized using the assignment p_info.link = & q_info. Note that this statement has the effect of sharing p_info.link rather than copying it. This is not an optimization, but is a necessary step to ensure that the transformation handles aliasing effects correctly.

2.3.3 Pointer dereferencing

The transformation inserts checks, implemented using macros CHECK_SPATIAL and CHECK_TEMPORAL, before pointer dereferences.

Original	Transformed
`x = * q;`	`CHECK_TEMPORAL(q_info);` `CHECK_SPATIAL(` `q, sizeof(*q), q_info);` `x = * q;`
`y = p->a->b;`	`CHECK_TEMPORAL(p_info);` `CHECK_SPATIAL(&(p->a),` `sizeof(p->a), p_info);` `CHECK_TEMPORAL(` `p_info.link->a_info);` `CHECK_SPATIAL(&(p->a->b),` `sizeof(p->a->b),` `p_info.link->a_info);` `y = p->a->b;`

2.3.4 Function calls

Functions that take pointer parameters are transformed so as to accept additional parameters that hold metadata pertaining to these parameters. In addition, the return value of a function that returns a pointer (or a structure containing a pointer) is modified so as to return the associated metadata.

Note that the introduction of additional parameters affects compatibility with external functions. Unlike fat pointers, this incompatibility can be easily fixed in most cases. In particular, if an external function takes objects containing pointer values but does not modify them (or return pointers), then calls to this function are left untransformed. If the function modifies pointer values or returns them, then wrapper functions will need to be created. Or alternatively we can just assume these pointers are always valid. The only drawback is that memory errors caused by these pointers will not be detected. In our experience, we have found that at least among the standard libraries used by our benchmark programs, it is rare for programs to use external functions that require wrapper functions.

3. HANDLING TYPE CASTS AND UNIONS

The C language allows programs to perform arbitrary casts. However, almost all casts in typical C programs involve types that have some sort of subtype—supertype relationship [32]. Such casts can be classified into upcasts (cast from subtype to supertype) and downcasts (casting from a supertype into a subtype). As in object-oriented languages, upcasts are always safe. But downcasts are not safe, and need to be checked at runtime by maintaining *runtime type information (RTTI)* with each object.

The following changes are made to the basic transformation technique to support casting between subtypes and supertypes:

- *Declarations:* The type and pointer variable declarations are modified to incorporate an additional metadata field called `tid` that identifies the actual type of data stored in the referent.
- *Casts:* Note that the info variables for all pointers have the same structure: they have the same layout and the same interpretation of the fields of this structure. Thus casts between any two pointer types are allowable, as long as the runtime type associated with the pointer is checked prior to the use of the `link` field of its info variable.
- *Dereferencing:* For `p->a`, the transformation is as before if `p->a` is a non-pointer. But if it is a pointer, then the info associated with `p->a` will be invalid if `p_info.tid` is NOT a subtype of the static type of `*p`.
- *Assignments:* For assignments between pointers, the `tid` field must also be copied on pointer assignments. When pointers are created using `malloc` or `&` operator, the `tid` field needs to be populated. For `malloc`, the type of the pointer is used for this purpose. For the `&` operator, the statically declared type of the object (whose address is being taken) is used.

Pointer arithmetic operations need some care. Consider a pointer arithmetic statement such as `p+=k`. Let `elem_sz` and `info_sz` be the element sizes of runtime type and info type of the object pointed by `p` respectively. (We maintain these sizes information in transformed programs.) Then the link field of `p_info` is also incremented, using `p_info.link=(void*)p_info.link+k'`, where `k'` equals `(k*sizeof(*p)/elem_sz)*info_sz)`. This statement increments `link` to point to the info variable for the new value of `*p`, if any. Note that this transformation supports programs which advance structure pointers by first casting them into `char*` or `void*`, then adding carefully calculated offsets to them and casting them back to pointers of desired types. To ensure that this will result in a correct link field value, we must ensure that the product of `k*sizeof(*p)` be a multiple of `elem_sz`. If this condition is not satisfied, the link field is marked invalid (by setting `p_info.tid` to a special value) and hence cannot be used. It is worth mentioning that our transformation still allows dereferencing of `p`, but causes problems if pointers contained within `*p` are accessed. Thus, it will be possible to write a program that takes a pointer to an array of structures, casts it into a `char*`, and sequentially accesses the entire array as a sequence of characters. However, problems do arise if pointer arithmetic is used to advance through an array that is embedded within a structure when the array elements contain pointers that are dereferenced. Although we have not encountered such situations in our experiments, we have begun developing an improved treatment of pointer arithmetic that overcomes this limitation.

3.1 Notion of Subtyping

When a type $T1$ is laid out in memory exactly as a prefix of another type $T2$, type $T2$ is called a *physical subtype* of type $T1$. Physical subtyping has been studied for understanding type casts in C programs [32], but it is unnecessarily restrictive for our purposes. To maximize the set of programs that can be handled by our approach, we define the following weaker notion of subtyping that is sufficient to ensure soundness of the approach.

Consider a program fragment:

```
T2 *q;
T1 *p = (T1*)malloc(....);
.... some code that initializes p ...
q = (T2 *)p;
.... code that dereferences q ....
```

Note that if `T2` contains no pointers, then this use is always acceptable. Any spatial or temporal error in accessing a field of the form `q->a` will be detected using the metadata in `q_info`, which would have been copied from `p_info`. On the other hand, if `T2` contains a pointer, say, in a field named `b`, then we need to make sure that correct metadata information is available for checking an access of the form `q->b->c`. Suppose that we blindly access `q_info.link->b_info`. Since `q_info.link` was assigned from `p_info.link`, this access will be equivalent to `p_info.link->d_info`, where `d_info` is at same offset in `T1_info` as is `b_info` in `T2_info`. Similarly, the `q->b` accesses the same location as `p->e` where `b` and `e` are at the same offset with respect to the base of `T2` and `T1` respectively. Thus, the access `q_info.link->b_info` will yield correct results if `p->e` is a pointer and `p_info.link->d_info` corresponds to the metadata associated with this pointer. Note that the types of pointers `p->e` and `q->b` need not be the same: if an access `q->b->c` is incorrect, it will be detected when the runtime type of `q_info.link->b_info.link` is examined.

Based on the above discussion, we define the following notion of subtyping. Suppose that `T1` contains pointer fields at offsets $p_1, ..., p_n$, where p_i's are in sorted order. Similarly, let $q_1, ..., q_m$ denote all of the offsets of pointers fields within `T2`. Now, `T1` is a subtype of `T2` iff $m \leq n$ and $p_i = q_i$ for $1 \leq i \leq m$.

3.2 Supporting Unions

Original	Additional
	`struct ptr_info {`
	`int tid;`
	`char *base;`
	`unsigned long size;`
	`capability *cap_ptr;`
	`unsigned long cap_index;`
	`struct ptr_info * link;`
	`};`
`union u {`	`union u_info {`
`char *p1;`	`struct ptr_info p1_info;`
`struct stu *p2;`	`struct {`
`};`	`int tid;`
	`char *base;`
	`unsigned long size;`
	`capability *cap_ptr;`
	`unsigned long cap_index;`
	`struct stu_info *link;`
	`} p2_info;`
	`};`

Unions in C represent implicit type casts when a union member is stored as one type and accessed as another. Thus, the basic machinery we have developed above will work for dealing with unions. The previous example shows how a union definition is transformed.

Note that the info type for the union does not have tags. This is not a problem, since the `tid` is always the first field of all the structs within `u_info`, and can hence be accessed to determine the type of the pointer currently stored in the union `u`. To ensure that this approach will work correctly, it is necessary to create and maintain the `tid` field even when the union holds a non-pointer. For instance, if an integer field `p3` is added to the above union, then a field `p3_info` will be added to `u_info`. When an integer is stored into the union using the field `p3`, then `p3_info.tid` will be set to indicate that the current content of the union is an integer.

It is important to note that since we do not convert unions into "tagged unions," as is the case in [28] or Cyclone [19], programs that expect to store into `p1` and access this later using the field name `p2` will work as expected. However, any attempt to dereference this pointer will be checked using the runtime type information, so invalid memory accesses will be caught.

4. OPTIMIZATIONS

4.1 Local Optimizations

The most significant optimization we have performed is to separate metadata into two data structures: a `header` that holds information relating to a memory block, and a `ptr_info` that holds information relating to a specific pointer. The following table illustrates this separation.

Original	Transformed
	`struct header {`
	`  void *base;`
	`  unsigned long size;`
	`  unsigned long cap_index;`
	`};`
`struct stu {`	`struct ptr_info {`
`  int  id;`	`  int tid;`
`  char *name;`	`  struct header *blkhdr;`
`};`	`  capability *cap_ptr;`
	`  struct ptr_info *link;`
`struct stu s;`	`};`
	`struct stu_info {`
	`  struct ptr_info name_info;`
	`};`
	`struct stu_info s_info;`
	`struct {`
	`  int tid;`
`struct stu *p;`	`  struct header *blkhdr;`
	`  capability *cap_ptr;`
	`  struct stu_info *link;`
	`} p_info;`

This separation yields significant savings when there are multiple pointers to the same block, since the header field in the block can be shared by all pointers to that block. Moreover, the split reduces the number of fields involved in maintenance operations, and hence improves performance.

Pointer dereferences tend to repeat frequently, and for each such dereference, our transformation generates pointer validity checks. However, `gcc` does not recognize and eliminate such checks when they involve fields in a structure. For the same reason, `gcc` optimizations cannot recognize and eliminate metadata updates even when the results of these updates are not used again. In order to exploit the common subexpression and dead variable elimination optimizations of `gcc`, we therefore convert metadata structures into individual variables whenever it is safe to do so. (In particular, we break the metadata structure associated with a local pointer variable when the address of the variable is not taken.) This optimization eliminates several redundant checks and updates at an intra-procedural level.

Another optimization we have implemented is to eliminate unnecessary operations on the stack capability store (SCS). This elimination is performed in the case of functions which never compute the address of a local variable, and hence have no risk of a local pointer escaping. This optimization primarily benefits function call-intensive programs where the frequently called functions do not contain reference operations that generate a pointer to a local variable.

The effect of these optimizations are discussed in Section 5.2.3. On the average, these optimizations improve performance by about a factor of two.

4.2 Global Optimizations

There are a number of global optimizations which can be done to eliminate checks and metadata, reducing the overhead of our approach from its current level. Unlike the previous optimizations, none of the optimizations discussed in this subsection have been implemented.

The first is to identify pointers whose type can be statically determined, and replace RTTI with constant type information for these pointers. This can be done using a fairly simple flow-insensitive type inference approach similar to the method used to determine so-called *safe pointers* in [26]. The results presented in [26] showed that for typical C programs, roughly 90% of pointers could be statically shown to be safe pointers, and similar results are likely for our approach. This is because safe and sequential pointers are never involved in (nontrivial) casts, and hence their static type will be the same as runtime type, there by making RTTI information redundant.

A second optimization is to identify pointer operations which can be statically determined to be temporally safe, and remove the temporal checks for these operations. Criteria for statically determining the temporal safety of a pointer access depend on the storage region(s) the pointer refers to. For heap pointers, there must be no path along which the referent could be freed between the previous check and the current dereference location. For stack pointers, there must be no chance that the pointer could have escaped from a previous function call. Static data pointers are inherently temporally safe.

A third optimization is to identify pointer operations which are statically spatially safe, and eliminate bounds checks for these pointers. The spatial safety of a pointer access is determined by examining all paths between the access and all recent checks/allocations. If the value of the current pointer expression is the same as the previous ones, then the access is spatially safe. If the value is increasing (e.g., the result of an increment operation such as `p++`), then the access is *lower-bound safe* but not *upper-bound safe*. Similarly, some accesses might be determined to be *upper-bound safe* but not *lower-bound safe*. This information can be used to eliminate or simplify checks inserted by the transformation.

Once checks have been eliminated, unnecessary maintenance code can be eliminated. Updates to metadata which can never reach a check can be detected using standard data flow techniques and eliminated. Metadata variables/fields which are never used can then be detected and eliminated. For pointers which exhibit partial safety (e.g., lower bound safety), the metadata can be simplified.

Program	Lines of code	Execution time Original (sec.)	Execution time Transformed Ratio	Peak heap usage Original (MB)	Peak heap usage Transformed Ratio	Executable size Original (KB)	Executable size Transformed Ratio
Olden							
bh	2080	1.37	2.82	0.17	4.76	81.7	1.97
bisort	684	0.66	1.76	1.57	5.51	30.6	1.43
em3d	561	2.77	1.79	6.53	3.13	44.7	1.14
health	709	1.08	2.72	2.33	5.45	50.9	1.21
mst	592	2.43	1.76	71.34	3.36	47.8	1.10
perimeter	395	1.49	3.37	2.74	4.72	29.6	1.04
power	763	1.81	1.22	0.42	3.12	43.8	1.54
treeadd	370	0.26	3.23	25.17	5.34	34.2	0.94
tsp	565	1.23	2.28	9.44	3.36	33.8	1.41
AVG			**2.33**		**4.31**		**1.31**
SPECINT							
099.go	29262	19.29	2.60	0.00	1.00	437.8	2.51
129.compress	1939	2.69	1.85	0.00	1.00	103.6	1.17
164.gzip	8620	1.21	1.46	6.61	1.08	166.5	1.34
175.vpr	17897	0.93	3.53	0.91	1.86	408.2	3.23
181.mcf	2413	0.20	2.85	96.59	3.01	129.5	1.10
AVG			**2.46**		**1.59**		**1.87**
Utilities							
bc-1.06	14264	2.59	2.69	0.09	2.67	191.1	2.82
gzip-1.2.4	8163	1.59	1.54	0.00	1.00	168.4	1.78
patch-2.5.4	11533	0.52	1.12	10.20	1.40	234.3	2.91
tar-1.12	24339	1.01	1.14	0.04	1.25	467.1	1.98
AVG			**1.62**		**1.58**		**2.37**

Figure 1: Transformation performance for Olden/SPECINT benchmarks and UNIX utilities. A ratio 1.22 means that the transformed program was 22% slower/larger than the original.

We believe that these optimizations will result in further significant improvements to the performance of our approach, and will be the primary focus of our future research in this area.

5. EXPERIMENTAL RESULTS

We have implemented a prototype of the transformation described in Sections 2 through 4. The prototype is a syntax-directed source-source transformation tool to instrument C programs. It uses CIL [25] as the front end and Objective Caml as the implementation language. C is a very complex language to deal with for program analysis and transformation. Our transformation work was considerably simplified by the intermediate language provided by CIL which consists of a clean subset of C constructs that can be easily manipulated and then emitted as C source code.

To evaluate the performance of our prototype implementation, we applied our transformation on a number of test programs from `Olden` [7], `SPECINT` [1] benchmarks, and UNIX utility programs. We used the `Olden` benchmarks included in the CCured package since the original benchmarks do not run on Linux. Although our implementation is not robust enough to handle all SPECINT programs, it is still able to process many moderate-sized programs — five of our test programs are over 10K lines of code.

Our current implementation does not support user-defined memory allocation/deallocation functions that are functionally different from `malloc`. Some of our benchmarks made use of such user-defined functions, so we had to modify these programs to replace the calls to these functions with standard functions such as `malloc` and `calloc`. No other source code modifications were required by our transformer.

5.1 Effectiveness

In our experiments, our implementation successfully detected several bugs which have been reported in previous research [26] in the SPECINT benchmarks: `compress` contains an array out-of-boundary error, and `go` contains several array out-of-boundary errors. In addition, our implementation detected an unreported array out-of-boundary bug in the UNIX utility program `patch`. All of these provide evidences to show that our implementation indeed works to identify memory access errors.

5.2 Performance

We compared the performance of original programs and transformed programs. Both of them were compiled using `gcc` version 3.2.2 with `-O2` optimization, and executed on an Intel Xeon 3.06GHz workstation with 3GB RAM running Red Hat Linux 9. All the execution times were measured using the total of system time and user time reported by `time`. Figure 1 shows the overall performance of our transformation.

5.2.1 Execution overhead

The third and fourth columns of Figure 1 are the comparison of execution time of original and transformed programs. For the `Olden` benchmarks, the transformed programs have a slowdown of 1.22 to 3.37, with an average of 2.33; for the `SPECINT` benchmarks, the slowdown ranges from 1.46 to 3.53, with an average of 2.46; for the UNIX utility programs, the range of slowdown is between 1.12 and 2.69, and the average is 1.62.

5.2.2 Memory overhead

The fifth and sixth columns of Figure 1 present the peak heap memory usage of both original and transformed programs. The main source of overhead in transformed programs is the space for storing `info` structures for the pointers that would be stored in each `malloc`'d block. Since there are four fields in the `info` structure associated with a pointer, the increase in memory usage can be 5 times in the worst case. Many `Olden` benchmarks use `malloc` primarily to allocate structures that contain mostly pointers. Thus, the memory overhead in these programs is close to this worst case possibility. For some programs, the increase exceeds a factor of 5 since there are other sources for memory overhead that also need to

Program	Transformed (ratio in execution time)		
	Spatial	Spatial & Temporal	Spatial & Temporal & RTTI
Olden			
bh	1.88	2.60	2.82
bisort	1.45	1.61	1.76
em3d	1.36	1.83	1.79
health	1.97	2.47	2.72
mst	1.32	1.59	1.76
perimeter	1.82	2.11	3.37
power	1.17	1.20	1.22
treeadd	1.81	2.73	3.23
tsp	1.88	1.81	2.28
AVG	*1.63*	*1.99*	*2.33*
SPECINT			
099.go	2.44	2.56	2.60
129.compress	1.37	1.65	1.85
164.gzip	1.15	1.33	1.46
175.vpr	2.05	2.88	3.53
181.mcf	2.85	2.50	2.85
AVG	*1.97*	*2.18*	*2.46*
Utilities			
bc-1.06	1.96	2.49	2.69
gzip-1.2.4	1.46	1.47	1.54
patch-2.5.4	1.06	1.06	1.12
tar-1.12	1.05	1.19	1.14
AVG	*1.38*	*1.55*	*1.62*
AVG	**1.67**	**1.95**	**2.21**

Figure 2: Performance overheads of checking different memory errors

Program	Transformed (ratio in execution time)			
	H/I merged	H/I sep.	H/I sep. SCS elim.	H/I sep. SCS elim. Info split
Olden				
bh	5.24	4.18	3.92	2.82
bisort	2.61	2.17	1.76	1.76
em3d	1.92	1.83	1.81	1.79
health	3.07	2.79	2.72	2.72
mst	2.28	2.06	1.77	1.76
perimeter	7.89	6.15	3.51	3.37
power	1.71	1.66	1.22	1.22
treeadd	4.92	4.12	3.96	3.23
tsp	4.44	3.06	2.28	2.28
AVG	*3.79*	*3.11*	*2.55*	*2.33*
SPECINT				
099.go	3.46	3.06	2.65	2.60
129.compress	5.32	3.82	2.11	1.85
164.gzip	3.03	2.38	1.54	1.46
175.vpr	4.49	3.99	3.67	3.53
181.mcf	5.60	3.90	2.90	2.85
AVG	*4.38*	*3.43*	*2.57*	*2.46*
Utilities				
bc-1.06	9.60	3.74	3.51	2.69
gzip-1.2.4	2.44	2.06	1.92	1.54
patch-2.5.4	1.27	1.13	1.12	1.12
tar-1.12	1.28	1.23	1.12	1.14
AVG	*3.65*	*2.04*	*1.92*	*1.62*
AVG	**3.92**	**2.96**	**2.42**	**2.21**

Figure 3: Effectiveness of optimizations

be considered. These include a fixed-length `header` for each heap memory allocation, and the memory used for HCS and SCS.

Most of the `SPECINT` benchmarks and the UNIX utilities we have tested do not store many pointers in heap, hence they do not increase heap memory usage significantly.

5.2.3 *Analysis of performance*

We first studied the performance overheads due to different kinds of checks inserted by our transformation: (a) checks for detecting spatial memory errors, (b) checks for detecting temporal memory errors, and (c) checks for verifying upcasts and downcasts using run-time type information. Columns 2-4 of Figure 2 show the normalized execution time for (a), (a)+(b), and (a)+(b)+(c) as compared to the original execution time of each program. The average execution time overheads for the three combinations are 67%, 95% and 121% for all the test programs.

We then studied the effectiveness of the optimizations. Previous research efforts on detecting both spatial and temporal errors have reported much higher runtime overheads than what is shown in Figure 1. Typical slowdowns are by about a factor of 4. In order to understand the source of performance improvement in our approach, we measured the execution time of our approach with each of the optimizations discussed in Section 4.

The second column in Figure 3 shows the slowdown when we combined the `header` and `info` metadata together. Note that the slowdown has now increased to about a factor of 4, which is consistent with the results reported by previous researchers. Column 3 of this table demonstrates that significant improvement in performance is obtained as a result of splitting the metadata into `header` and `info` variables. By eliminating unnecessary SCS operations, further significant decrease in overheads is obtained, as shown in the fourth column of the table. Finally, by converting the `info` structures into individual variables, we enable `gcc` to aggressively perform other optimizations. This factor leads to a further modest decrease in overhead. Together, these optimizations reduce the runtime overhead by about a factor of two.

The effectiveness of each optimization varies depending on the characteristics of the test programs. Programs that make a large number of function calls (e.g. perimeter) can benefit more from unnecessary SCS operation eliminations than programs that make fewer function calls (e.g. `health`). Programs that have a higher ratio of the number of pointer assignments over the number of pointer dereferences (e.g. `compress`) can have more performance improvement due to the separation of `header` and `info`.

5.2.4 *Performance comparison with related approaches*

Since the primary goal of our work is to detect both spatial and temporal errors, we limit our detailed comparison in this section to previous techniques that are capable of detecting both types of errors. It should be noted, however, that the comparison results should be interpreted carefully, since the set of benchmarks used in the related works are different from those used by us.

Patil and Fischer's shadow processing [28] is similar to our basic transformation, but has no support for downcasts. The approach checks all the temporal and spatial memory errors. Its overheads for a subset of the `SPECINT` benchmark programs is on an average of 538%, which is much higher than our result of 121%. The closely related approach Safe-C [2] also reports overheads over 300%.

The bounds-checking work of Jones and Kelly [20] provides better compatibility than our approach with untransformed external libraries. In particular, they do not require wrapper functions to be developed for such libraries, even if they return dynamically allocated data structures. On the downside, their approach incurs higher overheads, and moreover, does not detect dangling pointers to reallocated memory. They report a slowdown by a factor of 5 to 6 for most programs. Their work was initially distributed as an extension to `gcc 2.7`, and further work has continued on this approach, yielding gcc patches for subsequent versions of gcc. Most recent performance data for this method can be found in Ruwase and Lam [31], where they present an improvement to the original technique which has better compatibility with existing C programs. They report performance overheads that are virtually identical to that of the bounds-checking extension to gcc version 3.3.1. Their overheads for the common test programs such as gzip are about twice as much compared to our approach. In terms of worst case performance, they report slowdowns by a factor of more than 10, whereas the worst case slowdown reported for our work in Figure 1 is by a factor less than 4.

Yong and Horwitz [37] describe an approach which is similar to Purify [17] but is enhanced using global static analysis. They report an average overhead of 37% on the Olden benchmark. This is better than our result of 133%, but this improvement is obtained in their approach by checking pointer errors to write operations alone. Read operations, which far outnumber writes in typical programs, are not checked.

6. RELATED WORK

Much research effort has been put into the development of techniques for detecting memory access errors in C programs. We discuss works that are most closely related to ours.

6.1 Debugging-targeted approaches

These approaches typically operate by inserting checks into a program either at the source or binary level, with the goal of detecting certain subclasses of memory errors. These include Kendall's *bcc* [22], *Purify* [17], Steffen's *rtcc* [33], Loginov, et.al.'s work on *runtime type checking* [24], and Haugh and Bishop's STOBO tool [18]. Also in this category is the *CodeCenter* interpretive debugger [21]. Since these techniques are focused on debugging, performance is typically not a serious consideration. For instance, Purify and CodeCenter often have overheads in excess of 1000%. In contrast, our interest is in techniques that can be enabled in production code, so it is important to keep the overheads reasonably low. Another distinction is that our approach is focused on detecting all memory errors, while the above approaches generally target detection of a subset of errors that are commonly encountered. For instance, bcc and rtcc focus mainly on spatial errors. Similarly, Purify does not detect certain spatial errors (specifically, when pointer arithmetic causes a pointer to overshoot past the end of one object into the middle of the next object), or dangling pointers to reallocated memory. On the other hand, some of these approaches do provide better compatibility with external libraries, e.g., Purify can perform memory error checks within libraries that are provided in binary format.

6.2 Approaches that rely on garbage collection

Cyclone [19] is a variant of C that is designed with the express goal of reducing source code changes needed to convert C programs to Cyclone. However, it still makes significant changes to C-language syntax and semantics, e.g., C-style unions are replaced by ML-style unions. In addition, to support separate compilation, type declarations that provide detailed information about pointer types become necessary, and these declarations cannot be automatically generated. As a result, large C-programs require nontrivial effort to port to Cyclone. In contrast, our approach requires almost no changes to existing C-programs.

CCured, developed by Necula, McPeak, and Weimer [26], uses type-inferencing to differentiate pointers into *(definitely) safe* and *(potentially) unsafe* pointers at compile-time, and insert run-time checks to ensure validity of accesses through unsafe pointers. Unsafe pointers are further subdivided into those using pointer arithmetic (sequential pointers) and those using casts (dynamic pointers). The representation of safe pointers is unchanged, but a fat-pointer representation is used for unsafe pointers. This can cause some compatibility problems, as described in Section 2. In their recent work [9], they have separated metadata associated with sequential pointers to gain increased compatibility, especially with external libraries that often take pointers to arrays or character buffers. However, dynamic pointers continue to use fat pointer representation. To further reduce compatibility problems and to improve performance, they have added runtime type information, which can result in fewer pointers being classified as dynamic. However, the maintenance of RTTI is governed by programmer annotations, unlike our approach where it is automatic. In addition, their notion of subtyping is more restrictive than ours, so some programs that require the use of dynamic pointers can still continue to work with our notion of RTTI.

The most important difference between our approach and that of Cyclone or CCured is their reliance on garbage collection. While this change may be acceptable for some programs, it is not well-suited for applications such as OS kernels or other performance-critical or response-time critical servers.

6.3 Approaches targeting full compatibility with the C-runtime model

Safe-C inserts runtime checks to detect all memory errors. However, the approach introduces compatibility problems due to the use of fat pointers. These compatibility problems were addressed by Patil and Fischer [29, 28], but their performance overheads are still much higher than ours — by as much as a factor of 2 to 4 times over our approach. Moreover, neither of these approaches support downcasts, thereby significantly restricting the set of programs to which they can successfully be applied. The work of Jones and Kelly [20] is applicable to a much broader class of programs, including programs that cast pointers to integers and vice-versa. They use a *splay tree* at runtime to store information about every allocated memory chunk. Pointer validity is checked using this data structure before dereferencing. Unfortunately, the approach incurs heavy performance penalties (more than 10 times slowdown for pointer-intensive code), and hence cannot be used for production code. Moreover, it does not detect dangling pointers to reallocated memory. Another recent attempt is the *Fail-Safe ANSI-C Compiler* [27], which attempts to compile any ANSI-C program into a memory safe form, but is not yet complete.

The approach presented in this paper addresses the drawbacks of the above approaches, and provides a practical solution for detecting all memory errors with moderate overheads.

6.4 Security-targeted approaches

This category mainly consists of approaches that utilize code transformation to protect return addresses of activation record [12, 4, 8, 14], and for protecting against *format-string attacks* [10]. More recently, works have emerged that use randomization to achieve broad protection against attacks that exploit memory errors [5, 11]. While these techniques are very good in achieving their intended goal, namely, protecting against attacks launched by some outsider, and doing so with very low overheads, they are not useful for general-purpose memory error checking. In particular, these techniques do not detect access errors at all, but simply prevent such errors from being used to launch attacks.

Another security-targeted approach is CRED, developed by Ruwase and Lam [31]. The primary goals of CRED include better compatibility with existing C programs, and decreasing runtime overheads by focusing on memory errors that are most frequently associated with security attacks. CRED is based on Jones and Kelly's bounds-checking extension to `gcc`, and provides an option by which runtime overheads can be greatly reduced by restricting error checks to operations involving string buffers. In the full-checking mode, the performance of CRED is close to that of Jones and Kelly's. A more detailed discussion of this performance can be found in Section 5.2.4.

There have also been a number of approaches which rely on static source code analysis to detect potential security-related memory errors prior to execution [13, 16, 36, 23, 35, 30, 15], but these are not as powerful as runtime detection techniques and hence not used widely.

Several techniques which combine static analysis with dynamic checking have also emerged recently. Yong and Horwitz [37] presented an approach to detect invalid memory writes by combining static analysis and a run-time technique similar to Purify [17]. By reducing the size of checked memory regions, they achieved both good run-time performance and improved error detections. Their performance overheads are generally lower than ours. However, this result is achieved by omitting error checks for read operations, which are typically much larger in number than write operations. Moreover, because of its reliance on Purify-like techniques, it cannot detect all memory errors.

Lhee and Chapin [34] developed a technique called type-assisted dynamic buffer overflow detection, in which the compiler is extended to provide information about the sizes of buffers and library calls are intercepted. Their technique detects many errors with fairly low overhead, but like many other security-targeted approaches, focus only on array-based spatial errors that occur within a prespecified set of library calls. Dynamically sized automatic arrays (i.e., those allocated by the `alloca` function) are also not supported by their approach. Avijit, Gupta, and Gupta [3] invented a very similar approach called TIED/LibsafePlus which operates on binary (rather than source) code.

7. DISCUSSION AND CONCLUSION

The ultimate goal of memory error detection is to develop an approach which possesses four essential properties:

(1) the ability to detect all memory errors,

(2) the ability of handle all C programs without modification,

(3) low performance overhead, and

(4) preservation of the existing C memory allocation model.

The approach presented in this paper already satisfies properties (1) and (4), but still has some shortcomings with regard to the other areas. However, we believe that with some additional work, the approach can be extended to satisfy (2) and (3) as well. To satisfy (2), arbitrary casts must be handled. To satisfy (3), additional optimizations are required, which were discussed in section 4.2. We discuss support for arbitrary casting next, and follow with some additional potential extensions to make the approach more useful.

7.1 Supporting Arbitrary Casts

There are two kinds of typecasting operation that the current implementation does not support. The first is casting from an integer type to a pointer. The second is casting between structure pointers in a manner that violates the subtype criteria discussed in Section 3.

Integer to pointer casts can be handled by treating integers which can receive cast pointer values as pointers, i.e., an integer `i` that is assigned the result of a pointer expression cast to type integer has an associated `i_info` variable allocated by the transformation. Along paths which assign non-pointer values to `i`, the `base` field of `i_info` is set to `NULL`. Limiting the number of integers treated as pointers is critical to keep the runtime overhead reasonably low. A fairly simple flow-insensitive data flow analysis or type inference can be used to identify variables likely to be assigned converted pointer values at runtime.

A second type of cast that presents challenges is casting from one structure type to another, where the two structure types have pointer fields at different offsets. In this case, the casting operation is implicitly performing integer-to-pointer and/or pointer-to-integer casts on the values contained within the structure fields. These types of casts can be handled by treating the involved fields in a manner similar to integer variables which can receive pointer values; by treating the fields as if they are pointers, and allocating the corresponding `info` fields within the structure pointed by `link`. Such fields can be identified by the same analysis used to identify integer variables receiving pointer values.

7.2 Support for Non-Standard Memory Allocation

Some programs implement their own memory management functions on top of the C runtime library. Calls to program-defined allocation functions can be handled if the functions exhibit semantics similar to those of `malloc`/`free`. The transformation tool could be extended with an option in which the user can specify which functions are similar to `malloc` or `free`, then the transformation tool would ignore the internal behavior of these functions, and insert checking code which trusts the user-defined functions to behave correctly. This approach would catch any errors due to misuse of the user-defined allocation functions, but would not detect any bugs within the allocation functions themselves.

It is nontrivial in our current implementation to support customized memory management functions that have different semantics compared to `malloc`/`free`.

7.3 Summary

The source transformation presented in this paper dynamically detects memory errors for C programs which do not use customized memory management functions that are fundamentally different from `malloc` and obey two elementary casting restrictions: no casting of integers to pointers, and no casting between pointers to structure types which do not possess a subtype or supertype relationship. For such programs, the approach detects all spatial and temporal memory errors, without modifications to program source code, and without changing the memory allocation model. Moreover, when compared to previous approaches which detect temporal errors, our approach decreases overhead by at least a factor of two.

8. REFERENCES

[1] Anonymous. *SPEC CINT Benchmark*. Standard Performance Evaluation Corporation. http://www.specbench.org/.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 290–301, June 1994.

[3] K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: Tools for runtime buffer overflow protection. In *USENIX Security Symposium*, pages 45–55, 2004.

[4] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference*, pages 251–262, Berkeley, CA, June 2000.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, Washington, DC, August 2003.

[6] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. In *Software - Practice and Experience*, pages 807–820, 1988.

[7] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 29–38, Santa Barbara, CA, USA, 1995. ACM Press.

[8] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *International Conference on Distributed Computing Systems (ICDCS)*, April 2001.

[9] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 232–244, June 2003.

[10] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, 2001.

[11] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, Washington, D.C., August 2003.

[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Automatic detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, January 1998.

[13] N. Dor, M. Rodeh, and M. Sagiv. Cssv: Towards a realistic tool for statically detecting all buffer overflows in c. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2003.

[14] H. Etoh and K. Yoda. Protecting from stack-smashing attacks. Published on World-Wide Web, June 2000.

[15] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, GA, May 1999.

[16] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *ACM Conference on Computer and Communication Security (CCS)*, pages 345–354, 2003.

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

[18] E. Haugh and M. Bishop. Testing C programs for buffer overflow vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, February 2003.

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[20] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *International Workshop on Automated and Algorithmic Debugging*, pages 13–26, 1997.

[21] S. Kaufer, R. Lopez, and S. Pratap. Saber-C: an interpreter-based programming environment for the C language. In *Proceedings of the Summer USENIX Conference*, pages 161–171, 1988.

[22] S. C. Kendall. Bcc: run–time checking for c programs. In *Proceedings of the USENIX Summer Conference*, El. Cerrito, California, USA, 1983. USENIX Association.

[23] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, pages 177–190, 2001.

[24] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps. Debugging via run-time type checking. In *Fundamental Approaches to Software Engineering*, 2001.

[25] S. McPeak, G. C. Necula, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Conference on Compiler Construction*, pages 213–228, 2002.

[26] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 128–139, January 2002.

[27] Y. Oiwa, T. Sekiguchi, E. Sumii, and A. Yonezawa. Fail-safe ansi-c compiler: An approach to making c programs secure (progress report). In *International Symposium on Software Security*, number 2609 in LNCS, pages 133–153. Springer-Verlag, 2002.

[28] H. Patil and C. N. Fischer. Low-cost, concurrent checking of pointer and array accesses in c programs. *Software - Practice and Experience*, 27(1):87–110, 1997.

[29] H. G. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *International Workshop on Automated and Algorithmic Debugging*, 1995.

[30] R. Rugina and M. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 182–195. ACM Press, 2000.

[31] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Network and Distributed System Security Symposium (NDSS)*, pages 159–169, February 2004.

[32] M. Siff, S. Chandra, T. Ball, K. Kunchithapadam, and T. Reps. Coping with type casts in C. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 180–198. Springer-Verlag, 1999.

[33] J. L. Steffen. Adding run-time checking to the portable c compiler. *Software - Practice and Experience*, 22(4):305–316, April 1992.

[34] K. suk Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *USENIX Security Symposium*, pages 81–88, 2002.

[35] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2000.

[36] Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *European Software Engineering Conference / ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 327–336. ACM Press, 2003.

[37] S. H. Yong and S. Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, 2003.