Efficient Detection of All Pointer and Array Access Errors

Todd M. Austin, Scott E. Breach and Gurindar S. Sohi
University of Wisconsin-Madison

Presented by Wei Li

1 Introduction

A memory access error is any dereference of a pointer or subscripted array reference with reads or writes storage outside of the referent.

- spatial access error - outside of the address bounds of referent
- temporal access error - outside of the lifetime of the referent

Memory access errors are possible in languages with arrays, pointers, local references, or explicit dynamic storage management. Such errors are difficult to detect and fix because:

- The effect may not manifest themselves except under exceptional conditions
- The exceptional conditions may be very difficult to reproduce
- It is difficult to think the program related to memory access errors

So, we would like to the program execution environment to support memory access protection at the variable level. The idea would be at compile-time, to use an extended representation which we call a safe pointer. A safe pointer contains the value of the pointer as well as object attributes. The object attributes describe the location, size and lifetime of the pointer referent. As the pointer is manipulated by some pointer operations, the object attributes are transferred to any new safe pointer values.

2 Safe Pointers

To enforce access protection, we must extend the notion of a pointer value to include information about the referent. The idea of safe pointer originated from the tagged pointers in many Lisp implementations.

The following code gives the C-like definition for safe pointer.
typedef {
    <type> *value ;
    <type> *base ;
    unsigned size ;
    enum {Heap=0, Local, Global} storageClass ;
    int capability ; /* plus FOREVER and NEVER */
} SafePtr<type> ;

The use of the attributes are described here:

value:  The value of the safe pointer; it must contain any expressible address.

base:  and size:  The base address of the referent and its size in bytes.

storageClass:  The allocation type: Heap, Local or Global.

capability:  When dynamic variables are created, a unique capability is
issued to that storage allocation. The unique capability is also inserted
into an associated store called the capability store and deleted from
that store when the dynamic storage allocation is freed or when the
procedure invocation returns.

The value attribute is only safe pointer member that can be manip-
ulated by the program source.  base and size are the spatial attributes.
storageClass and capability are the temporal attributes.

Safe pointers can exist in three states: safe, invalid and valid. Only valid
pointers can be dereferenced.

3  Program Transformation

3.1  Pointer Conversion  All pointer definitions and declarations must be
extended to include object attributes.

3.2  Check Insertion  The dereference check first verifies that the referent
is alive in the capability store. Then a bound check is applied to verify
that the entire extent of the access fits into the referent.

3.3  Operator Conversion  If the operator creates a new pointer value,
it must include an unmodified copy of the pointer operand's object
attribute. To handle the reference operator, e.g., the '&' operator, we
decompose access paths into two parts, a prefix and suffix.
3.4 Run-Time Support Run time memory functions in C like 'malloc',
'free' etc, function invocation entry and exit point have to be rewritten
to detect memory leakage.

4 Optimizing Dereference Checks

To optimize and to minimize the checks in the program, we can elide some
dereference checks with the following rule:

A check at a dereference of pointer value v may be elided at program
point p if the previous equivalent check executed on v has not been
invalidated by some program action.

Run-Time Check Optimization Spatial checks have no side effects. So
we store the operands to the last check in the safe pointer object at-
tributes, which amounts to the effective address of the last dereference.
To elide temporal checks, we keep a copy of a global copy counter, in-
ccremented when storage is deallocated, in the safe pointer. If this safe
counter, has not changed since the last temporal check, the referent
has not been freed and the temporal check can be safely elided.

Compile-time Check Optimization The algorithm implements a for-
ward data-flow framework similar to that used by common subex-
pression elimination. The algorithm executes in these stages:

i The algorithm seeds data-flow analysis by approximating all out
sets.

ii The data-flow framework is solved to determine where check ex-
pressions reach in the program.

iii The in sets are used to elide redundant checks. Checks may be
elided wherever a lexically identical or equivalent check is avail-
able in the block.

We must also be aware that kills may occur through function calls or aliases.
In either cases, we must make a conservative approximation if insufficient
information is available and assume that a kill does occur.

5 Conclusion

In this paper, we presented a pointer and array access checking technique
that provides complete error coverage through a simple set of program trans-
formations. It is the first technique that detects all spatial and temporal access errors. It can be applied to compiled or interpreted language with subscripted and mutable pointers, local references, unions, and explicit and type-less dynamic storage management. All of these are directly related to computer security in that pointer operation are heavily used by malicious users to attack the system.

From the experiment results, the performance with only run-time resolved optimization was quite good. For all tested programs, instruction execution overheads ranged from 130typically below 100safe programs are the lack of safe pointer register allocation and ineffective optimization in the presence of check function. The solution to these problems will better the integration between the safe compiler and the code generation.