8-2014

Performance of Compiler-Assisted Memory Safety Checking

David Keaton
Carnegie Mellon University, dmk@cert.org

Robert Seacord
Carnegie Mellon University, rcs@sei.cmu.edu

Software Engineering Institute

Performance of Compiler-Assisted Memory Safety Checking

David Keaton
Robert C. Seacord

August 2014

TECHNICAL NOTE
CMU/SEI-2014-TN-014

CERT Division

http://www.sei.cmu.edu

Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0001495

Table of Contents

List of Figures

List of Tables

Abstract

Buffer overflows affect a large installed base of C code. This technical note describes the criteria for deploying a compiler-based memory safety checking tool and the performance that can be achieved with two such tools whose source code is freely available. The note then describes a modification to the LLVM compiler to enable hoisting bounds checks from loops and functions. This proof-of-concept prototype has been used to demonstrate how these optimizations can be performed reliably on bounds checks to improve their performance. However, the performance of bounds propagation is the dominant cost, and the overall runtime cost for bounds checking for C remains expensive, even after these optimizations are applied. Nevertheless, optimized bounds checks are adequate for non-performance-critical applications, and improvements in processor technology may allow optimized bounds checking to be used with performance-critical applications.

1 Introduction

Buffer overflow is the leading cause of software security vulnerabilities. It is responsible for 14% of all vulnerabilities and 35% of critical vulnerabilities (Common Vulnerability Scoring System score of 10) over the past 25 years, as reported by Sourcefire [Younan 2013].

The C programming language provides a powerful set of low-level systems programming features to software developers, which, if misused, can result in buffer overflows. Features such as pointer arithmetic, pointers that can point to a location other than the beginning of an object, and the definition of array accesses as pointer dereferences make it difficult to determine if a memory access is in bounds.

As of July 2014, the TIOBE index shows C as the most popular language with 17.1% of the market. Because of its simplicity and transparent performance, C is still heavily relied upon by embedded systems, network stacks, networked applications, and high-performance computing. Embedded systems can be especially vulnerable to buffer overflows because many of them lack hardware memory management units. Network software is frequently stressed both by heavy use in unpredictable environments and by attacks.

For greenfield software development projects (developed without constraints from previous systems), it may be practical to mandate a language subset or annotations, or to specify a different language altogether. However, because most development efforts are extensions or modifications of existing code, it is necessary to find a solution to eliminating buffer overflows that can work with legacy code.

This necessity introduces additional constraints, for example, that the mechanism should not require changes to the source code. Because developers often do not have control over the complete system on which an application will run, application binary interface (ABI) compatibility should also be maintained. Finally, for the same reason, it should be possible to link checked code with unchecked binary libraries for which the source code might not be available.

We measured the performance of two memory safety checkers that meet these criteria and added an optimization that improved performance. In this technical note, we present the results and provide insight into some of the determining factors of memory checking performance.

Section 2 provides background, and Section 3 explains how we obtained our results. Results are presented in Section 4. Section 5 describes some related work, and Section 6 discusses future work. We conclude in Section 7.

2 Background

2.1 Definitions

When assigning a value to a pointer in C, the software developer has in mind a particular object to which the pointer should point, the *intended referent*. A memory access error occurs when an access through a pointer strays outside the bounds of the intended referent for that pointer. Because array accesses are pointer dereferences in C, the same failure mechanism also applies to arrays.

Spatial errors involve accessing an address outside the range of the intended referent. For example, an array index may be calculated using an incorrect formula, and as a result, accesses can occur outside the intended array. This can result in accessing unused memory or an object other than the one intended. A spatial error may result when a pointer contains an out-of-range value, or when the pointer's value is in range but the length of the access extends beyond the end of the intended referent.

Temporal errors result from attempting to access an object after the end of its lifetime. For example, a pointer to a stack object may be stored in the heap. If the function invocation containing the object returns and another function is called, dereferencing the pointer could access unused stack memory or an address within the new function's activation record. Any pointer that has a lifetime that extends outside the lifetime of its intended referent is a potential source of temporal errors.

Sometimes it is convenient to think of a pointer as *pointing to* an address. By this we mean that the value contained in the pointer is that address.

2.2 Valid Objects

Compiler-based memory error detection is more likely than other methods to have the information to track the intended referent. However, not all compiler-based strategies do so. One memory checking strategy is to maintain a runtime map of addresses where valid objects reside. The compiler instruments each pointer dereference to check that a valid address is being accessed.

A map of valid addresses could be as simple as a data structure containing address ranges for the stack, heap, and static data. This approach can catch some wild pointers but is imprecise because it allows accesses to bookkeeping information within the stack and heap in addition to program objects.

To increase precision, the compiler could generate code to build a table containing an entry for each valid object that might be accessed through a pointer. Statically allocated objects could be entered into the table at compile time, at link time, or at runtime before `main` executes. Heap allocation and deallocation routines could manage the entries for heap-based objects, and entries for stack-based objects could be managed by instrumentation that the compiler would add to the generated code.

Mudflap, which was built into versions of the GNU Compiler Collection (GCC) prior to 4.9, is an example of a memory-error detection mechanism that functions in this manner [Eigler 2003]. The

compiler instruments the code to register and unregister objects in an object database, which is implemented as a binary tree constructed at runtime.

Address Sanitizer, which is built into GCC and Clang, works in a similar fashion but implements the object database using a compressed shadow memory [Serebryany 2012]. Each location in program memory has a corresponding entry in shadow memory indicating whether or not that location contains a valid object. This mechanism does not require that the object table provide information about the beginning and end of each object. When an object is created, all the bytes it contains are marked valid, and when an object is destroyed, all its bytes are marked invalid.

These types of tools provide useful information and can catch many memory error conditions such as array accesses walking linearly off the end of an array. However, they may have false negatives because they track whether a memory address is within any valid object rather than tracking whether it is within the intended referent of the pointer being dereferenced.

2.3 Intended Referent via Object Tables

One memory safety checking method that tracks the intended referent begins with maintaining a table of valid objects. As with the mechanism discussed previously, the compiler instruments the code to check pointer dereferences against the object table. However, in this case, the object table must keep track of the beginning and end of each object.

Figure 1 illustrates object table organization. The table is indexed by the address range of an object. The address contained in a pointer is used as the lookup key to find a match to an object's address range, and the table lookup yields the base and limit addresses of the object. Alternatively, an object table could contain the base address and the size of the object.

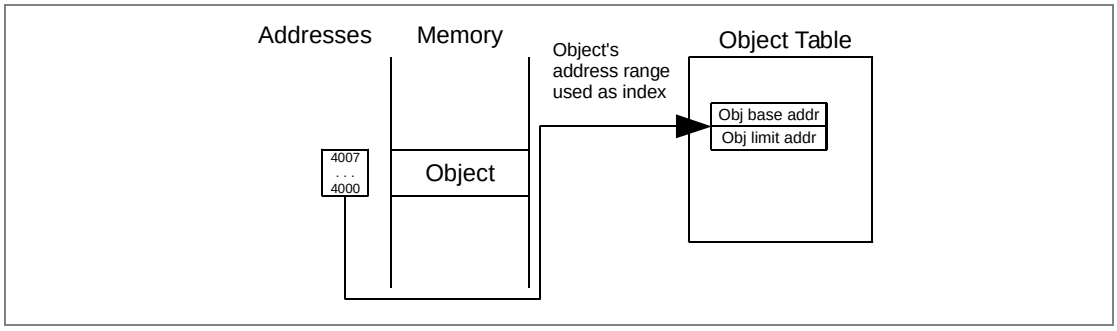

Figure 1: Object Table Organization

Pointer arithmetic may result in an address that is outside the intended referent but falls within another valid object described in the object table. Dereferencing the result will appear to be valid if the intended referent is not tracked. Therefore, to avoid false negatives, there must be some assurance that no pointer strays outside the bounds of its intended referent. To accomplish this, the code is instrumented to ensure that the result of a pointer arithmetic operation refers to the same object to which the original pointer referred before the arithmetic took place. For example, consider the following code containing pointer arithmetic.

```
int *p, *q;

/* . . . */
q = p + 1;
```

The instrumented code looks up the address contained in `p` in the object table to determine which object it designates. The base and limit of that object are then compared against the result of the expression `p + 1` to determine if the new pointer value is still within the same object.

If `p` points within an object but `p + 1` does not point within the same object, an invalid address has been generated. If `p` does not point within any known object, then it may point to an unchecked object. This could occur if the object originated from an unchecked binary library. This is a source of unsoundness, but checked and unchecked objects can at least be distinguished from each other. In this case, `p + 1` also must not point within any known object; otherwise an invalid address has been generated.

When the results of the pointer arithmetic are known to have exceeded the bounds of the intended referent, either by straying outside of the original known object or by wandering into a known object from an unchecked location, the expression returns a special value indicating that the address is invalid. If the pointer is dereferenced later, the dereference check will notice the invalid pointer and report an error.

This mechanism does not protect against an uninitialized pointer, which may accidentally point within a known object or may point to an unchecked object. Combining this method with other techniques, such as initializing to zero each pointer that lacks an initializer, can eliminate this problem.

The C programming language is somewhat unique in that a C program is allowed to compute the next address past the end of an object as long as it does not attempt to dereference that address. As a result, when checking pointer arithmetic, this method must allow the computation of the address one past the end of an object. When checking dereferences, one past the end is invalid.

This memory safety checking method was introduced by Jones and Kelly [Jones 1997]. It originally added padding to each object so that the address one past the end was not part of any other object, which required special workarounds for function parameters to avoid breaking the ABI. In the Jones and Kelly method, once a pointer strays outside the bounds of an object, it is "stuck" at the value that indicates it is invalid.

Ruwase and Lam refined the process by creating a new descriptor for each out-of-bounds (OOB) pointer value [Ruwase 2004]. They called the descriptor an *OOB object*. It contains the result of the offending pointer arithmetic and identifies the object to which it is intended to refer. The address of the OOB object is then stored in the pointer where it is available for checking during future pointer arithmetic and dereferences. An advantage of this approach is that it can handle real-world code that calculates an address in stages, where an intermediate stage might be out of bounds but the final result is in bounds. Although this is undefined behavior in C and is also a violation of *The CERT C Coding Standard* rule "ARR30-C. Do not form or use out-of-bounds pointers or array subscripts" [Seacord 2014], Ruwase and Lam discovered such operations in enough actual code that they found it useful to allow them. A disadvantage of this approach is that if a pointer containing the address of an OOB object is passed to unchecked code, the unchecked code may store through that pointer, which would damage the metadata contained in the OOB object itself and therefore compromise the memory safety checking mechanism.

Dhurjati and Adve added several optimizations [Dhurjati 2006a]. One of their changes was to set an out-of-bounds pointer to an inaccessible address that would trap when dereferenced instead of using the address of the OOB object. They attempted to use this technique to eliminate checks on dereferences, because the trap would do the work for them. Unfortunately, this scheme fails to detect buffer overflows that occur as a result of a misaligned pointer that points to a location near the end of the buffer. In these cases, the beginning of the access is in bounds, but the end is out of bounds. The misalignment might be accidental or intentional. For example, intentional misalignment occurs when networking code packs data items to reduce packet size and to exchange information portably among systems with different ABIs. Some processors are unable to perform misaligned accesses, and in those cases the software must perform the packing and unpacking explicitly. On processors that can access misaligned data directly, the software may make use of that hardware facility for performance. On such processors, using trap values for out-of-bounds addresses does not eliminate the need for dereference checks. However, it is still advantageous to implement OOB pointer values as trapping addresses, even if dereferences are checked, because then passing an OOB pointer to unchecked code will lead to a trap rather than overwriting the metadata.

The Dhurjati and Adve approach was implemented in SAFECode [Dhurjati 2006b]. A second version of SAFECode, which adds dereference checks, is available for use with recent versions of the Clang compiler. This second version of SAFECode was selected as one of the memory safety checkers examined in this study.

Plum and Keaton presented additional work along these lines which included caching the base and bounds of the intended referent for certain kinds of pointers, as well as a static analysis method to optimize away some of the checks [Plum 2005]. The work presented in this technical note adapted part of the analysis as described in Section 3.2.

2.4 Intended Referent via Pointer Tables

Another method for tracking the intended referent is to associate the bounds information with each pointer rather than with each object. When a pointer is dereferenced, it is compared against its bounds to determine if the memory access is valid.

Bounds information could be associated with pointers by increasing the size of a pointer to include its current value, base address, and limit address or size. However, doing so would change the ABI and would therefore be impractical if the application developer does not have control over the complete environment. In addition, a large amount of software is written with hard-coded dependencies on the sizes of pointers and other objects.

A solution is to maintain a table with an entry for each pointer and store the bounds of the intended referent in the table. Figure 2 illustrates pointer table organization. The address of the pointer, rather than the value contained in the pointer, is the lookup key that is used to find the base and limit addresses of the intended referent.

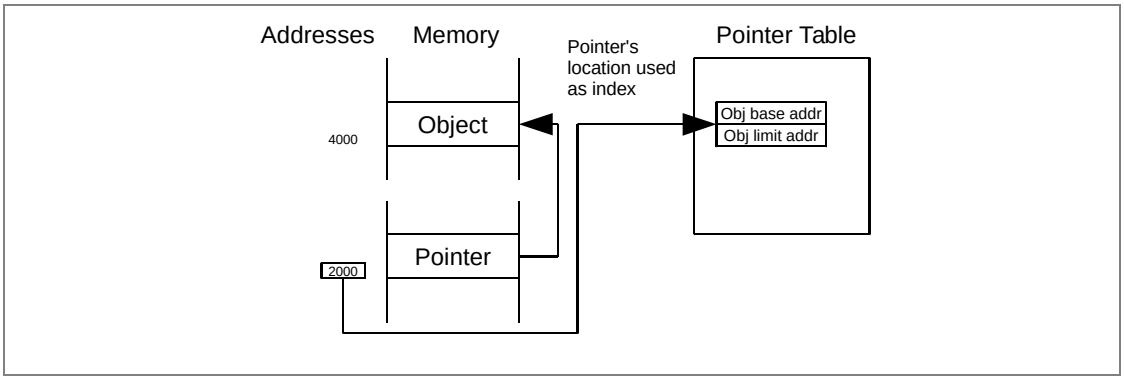

Figure 2: Pointer Table Organization

When a pointer is located only in registers and is never written to memory, it does not have an address to use as a key for pointer table lookups. In that case, the compiler could allocate memory space for the pointer anyway, acting as if the pointer variable had its address taken, to generate an address to use as a unique lookup key.

Alternatively, for pointers that are never stored in memory, the compiler can create additional local variables to hold the base and limit, and avoid storing them in the pointer table. This decreases the size of the table and reduces the number of lookups. Taking this approach a step further, even if a pointer is sometimes written to memory, as long as its address is never taken so that all accesses to the pointer are clearly visible and unambiguous, the compiler can create a base and a limit variable with the same storage duration as the pointer and avoid the pointer table.

Pointer assignments are instrumented to copy the bounds of the source pointer to the bounds of the destination pointer. The following code illustrates an example.

```
int *p, *q;

/* . . . */
q = p + 1;
```

The expression on the right side of the equals sign is based on `p`. Therefore, when the value of the expression is assigned to `q`, the bounds of `p` are also copied to the bounds of `q`.

Bounds must also be propagated across function calls. This can be performed in a variety of ways, such as by adding extra parameters to checked functions to carry the base and limit of pointer arguments or by storing the bounds in an alternate stack alongside the regular program stack.

In the object table method, pointer arithmetic is checked because the pointer's value doubles as both an address for a future dereference and an indicator of the intended referent (via its entry in the object table, with the pointer value used as the lookup key). If a pointer calculation results in an address outside the bounds of the intended referent, then without checks on pointer arithmetic, the information about the intended referent would be lost. In contrast, with pointer tables, a pointer's value performs only one function, indicating the address for a future dereference. There is one set of bounds per individual pointer rather than per object, so the information about the intended referent is not lost even if a calculation results in an out-of-bounds pointer value. Consequently, an advantage of the pointer table method is that pointer arithmetic does not need to be checked.

Moreover, the pointer table method does not need to make any special accommodation for pointers that point one past the end of an object. Because only dereferences are checked, one past the end is treated the same as any other address that is outside the object. Any attempt to dereference an out-of-bounds address results in an error.

Another advantage of the pointer table method is that it can represent subobjects and suballocations. Consider the following structure.

```
struct {
  char a[8];
  int b;
} s;
```

Because the object table method associates the bounds with an object, there is one entry for the object `s` in the table. The structure `s` and the array `s.a` are represented by the same entry, with the bounds of `s`. The pointer table method associates the bounds with each pointer, so a pointer that points to `s.a` can have tighter bounds than one pointing to `s`, reflecting the subobject. By making appropriate annotations or intrinsic functions available, a pointer table system can also allow the software developer to narrow the bounds of a pointer explicitly to perform suballocations.

The pointer table approach was implemented by SoftBound+CETS (Compiler Enforced Temporal Safety for C) [Nagarakatte 2009], which is available for Clang. It is bundled with SAFECode to provide a choice of memory protection mechanisms in one package.

Unlike other approaches, SoftBound+CETS implements full temporal checking in addition to spatial checking. For simplicity, we focused on spatial checking for our experiment, as implemented by SoftBound without CETS. We performed our investigation using the latest version of SoftBound as bundled with SAFECode.

3 Methodology

3.1 Initial Performance

As a first step, we evaluated the performance of the existing SAFECode and SoftBound implementations using publicly available source code. These results may vary from previous results due to running in different environments and the possibility of modifications to the code occurring after any previous measurements. In addition, we made some changes to SAFECode and SoftBound as described in this section.

We chose the following benchmarks from SPEC CPU2006 because they are written in C and compile cleanly with Clang/LLVM 3.2, the latest version for which both SAFECode and SoftBound were available.

401.bzip2	433.milc
458.sjeng	470.lbm
464.h264ref	

As distributed at the time of this study, SAFECode added the LLVM `readonly` and `readnone` attributes to several of the checking functions to which it generated calls. However, at the time that the function calls are examined by optimizations, the calls do not return any results that are used elsewhere. Consequently, the calls to the checking functions were removed by the dead code elimination passes of LLVM. We removed the attributes to ensure correct memory safety checking before measuring the performance.

As distributed at the time of this study, SoftBound checked accesses to scalar objects, but it did not check calls to built-in functions such as `llvm.memcpy`, which access an array of objects. We added support for checking all such LLVM `MemIntrinsic` calls before measuring the performance.

All performance measurements were made at the `-O3` optimization level.

3.2 Performance Enhancements

We made three changes to SoftBound to investigate their effects on performance.

First, we hoisted spatial memory access checks out of loops when the loop bounds were known on entry. As an example, consider the following function.

```
#include <stddef.h>

void foo(int *a)
{
  for (size_t i = 0; i < 100; ++i)
    a[i] = i;
}
```

Figure 3 shows the generated LLVM code, including the effect of hoisting the check of the store to `a[i]` out of the loop. The optimization removes the struck-out text preceded by minus signs in the figure and inserts the bold text preceded by plus signs. The beginning of the check is adjusted to be the first element accessed (the beginning of array `a`), and the length of the check is adjusted to include all accesses that will be performed by the loop (400 bytes), rather than checking each iteration separately.

```
define void @foo(i32* nocapture %a) nounwind uwtable ssp {
entry:
  %0 = tail call i8* @__softboundcets_load_base_shadow_stack(i32 1) noun-
wind
  %1 = tail call i8* @__softboundcets_load_bound_shadow_stack(i32 1) noun-
wind
+ %bitcast = bitcast i32* %a to i8*
+ tail call void @__softboundcets_spatial_store_dereference_check(i8* %0,
i8* %1, i8* %bitcast, i64 400) nounwind
  br label %for.body

for.body:                                         ; preds = %for.body, %en-
try
  %i.04 = phi i64 [ 0, %entry ], [ %inc, %for.body ]
  %conv = trunc i64 %i.04 to i32
  %arrayidx = getelementptr inbounds i32* %a, i64 %i.04
- %bitcast = bitcast i32* %arrayidx to i8*
- tail call void @__softboundcets_spatial_store_dereference_check(i8* %0,
i8* %1, i8* %bitcast, i64 4) nounwind
  store i32 %conv, i32* %arrayidx, align 4, !tbaa !0
  %inc = add i64 %i.04, 1
  %exitcond = icmp eq i64 %inc, 100
  br i1 %exitcond, label %for.end, label %for.body

for.end:                                          ; preds = %for.body
  ret void
}
```

Figure 3: LLVM Code Showing the Effect of Hoisting a Bounds Check Out of a Loop

To prevent spurious error reports, with the check being executed and the offending access not executed, the check is hoisted only if it postdominates the first basic block inside the loop. It is possible to improve on this technique. For example, our implementation misses the opportunity to hoist the checks in the following code fragment.

```
for (size_t i = 0; i < 100; ++i)
  if (some_condition)
    a[i] = expression1;
  else
    a[i] = expression2;
```

Second, we hoisted bounds checks out of a function and into its callers when we could see all calls to the function, so that a bounds check will be executed somewhere (if necessary) if it is deleted from the original function. To see how this might be beneficial, consider the following program.

```
#include <stdio.h>
#include <stdlib.h>

static void foo(unsigned char *a)
{
  for (size_t i = 0; i < 100; ++i)
    a[i] = i;
}

int main(void)
{
  unsigned char a[100];

  foo(a);
  for (size_t i = 0; i < 100; ++i)
    printf(" %d", a[i]);
  putchar('\n');

  return EXIT_SUCCESS;
}
```

If the bounds check for the access to `a[i]` is first hoisted outside the loop in `foo` and then up to `main`, it is now in a position where the size of array `a` is known. Because the length of the bounds check is also known, it can be compared against the size of `a` to determine if the bounds check can be eliminated entirely. This occurs in the case of the preceding program. If, after hoisting the bounds check into the caller, there still is not enough information to eliminate the bounds check, it is performed within the caller, in hopes that it can still be eliminated along other call paths to the original function.

The mechanism used to accomplish this is to treat a bounds check in the called function as a precondition for that function (called a *requirement* by Plum and Keaton [Plum 2005]), in this case the precondition that the memory space pointed to by `a` is at least 400 bytes long. Then all requirements are checked at their call sites to see whether the bounds checks can be eliminated for that call path or merely hoisted out of the called function.

Inlining can accomplish the same thing by effectively hoisting a bounds check into the calling function, where there might be enough information to eliminate the check. Therefore, this mechanism provides a benefit in cases where inlining is not performed, such as when the called function is large.

SoftBound is implemented so that it operates on optimized code. First the optimizations are run, then SoftBound is run, and then the optimizations are repeated in an attempt to improve any code added by SoftBound. We found that unrolling loops thwarted some of our attempts to hoist bounds checks. Fully unrolled loops contained a sequence of memory accesses in straight-line code in place of one access within a loop. We therefore disabled loop unrolling. Alternative approaches would have been to disable it only for the first pass of optimizations or to write an additional optimization to combine the adjacent bounds checks that result from unrolling loops.

Our third change was to test the performance of bounds checks on stores only (to prevent arbitrary code execution), or on strings only (because incorrect string management is a leading cause of vulnerabilities), or only on stores to strings. Limiting the bounds checks in this way can provide some insight into the tradeoff between security and performance.

3.3 Performance Characteristics

We also measured the performance of SAFECode and SoftBound with checking turned off, to discover the performance effect of the maintenance and propagation of metadata via the object table maintained by SAFECode or the pointer table maintained by SoftBound. To accomplish this, we disabled the portions of SAFECode that emit pointer arithmetic checks and bounds checks, and we disabled the portions of SoftBound that emit bounds checks, leaving us with the bookkeeping code only and thereby establishing a ceiling for the performance benefit of bounds check optimizations.

4 Results

4.1 Initial Performance

Table 1 shows the baseline slowdown measured for SAFECode and SoftBound. All slowdowns shown in this technical note are measured relative to the Clang 3.2 compiler without bounds checking. SAFECode performance was reduced on average by a factor of 41.72 times, and the average slowdown for SoftBound was 5.36 (that is, a program runs 5.36 times slower with bounds checking than without it). It should be noted that SAFECode is in the process of being rewritten. The current rewrite produced the second version of SAFECode, which was used in this study, but work is still in progress and has so far focused on robustness and usability rather than performance [Criswell 2011]. It is expected that performance can be improved substantially in the future. Until then, most of our performance analysis is best performed using SoftBound.

Table 1: Baseline Slowdowns for SAFECode and SoftBound

	SAFECode	**SoftBound**
401.bzip2	32.17	3.45
458.sjeng	31.47	3.28
464.h264ref	60.54	7.95
433.milc	37.69	5.25
470.lbm	12.23	1.86
Average	41.72	5.36

4.2 Performance Enhancements

The first two bars of each group in Figure 4 show the slowdown for SoftBound plus hoisting bounds checks out of loops, and hoisting out of both loops and functions.

As shown in the figure, hoisting bounds checks out of functions did not provide a measurable benefit for our sample set. Our optimization found only three instances of bounds checks that could be hoisted out of functions, which was not enough to impact performance.

This outcome is partly due to inlining having already done much of the work and partly a result of performing the entire optimization at compile time. Plum and Keaton describe a method for performing some of the work at link time when information about all functions is available at once [Plum 2005].

The average slowdown of 4.72 shows that hoisting bounds checks out of loops provides a benefit compared with the slowdown of 5.36 without the optimization. The remaining three bars show the slowdown for performing bounds checks only on stores, only on strings, and only on stores to strings, in addition to hoisting bounds checks out of loops and functions. All three of these cases provide a performance benefit. The three are similar in magnitude, indicating that a useful tradeoff between performance and security may be achieved by checking only stores, with an average slowdown of 2.58. Security-focused applications would not be able to forego checking loads, however, because information leaks such as the Heartbleed bug would not be detected.

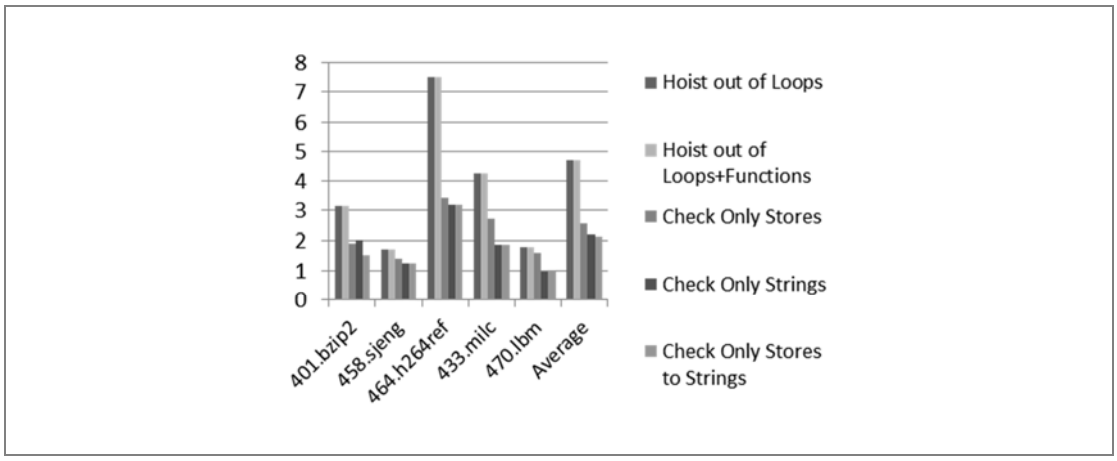

Figure 4: SoftBound Slowdown

4.3 Performance Characteristics

An average slowdown of 2.58 is reasonable for many situations but is too high for use in production mode for performance-sensitive applications. We investigated the performance breakdown of SAFECode and SoftBound to gain more insight into where time is spent. Table 2 shows the results of turning off checking to reveal the bookkeeping overhead.

Table 2: Slowdown of SAFECode and Softbound Metadata Maintenance and Propagation Only, Without Bounds Checks

	SAFECode Metadata Only	SoftBound Metadata Only
401.bzip2	1.00	1.25
458.sjeng	4.17	1.23
464.h264ref	5.62	3.15
433.milc	1.01	1.88
470.lbm	1.00	0.97
Average	3.02	2.07

The most striking result in comparison with the initial performance measurements in Table 1 is that SAFECode has similar bookkeeping overhead to SoftBound, with an average slowdown of 3.02 compared to SoftBound's 2.07. In some cases, SAFECode's bookkeeping overhead is too small even to be measurable, which suggests that when the SAFECode project begins to focus on performance, the most benefit will be gained by concentrating on pointer arithmetic checks and bounds checks.

Another interesting result is that hoisting bounds checks out of loops and checking only stores, which was shown to have a slowdown of 2.58 for SoftBound, is close to the slowdown of 2.07 for bookkeeping alone. This result indicates that for SoftBound, the most attention should be paid to improving the speed of metadata bookkeeping.

Table 3 is a profile of the benchmark with the most slowdown, 464.h264ref, running with SoftBound's metadata bookkeeping only. The top 15 functions are shown. It can be seen from the profile that SoftBound's shadow stack manipulation, its method for propagating bounds data across

function boundaries, takes 32.44% of the time in the benchmark. Metadata loads and stores take another 16.33% of the time.

Table 3: Profile of Top 15 Functions when Running 464.h264ref with SoftBound Metadata Bookkeeping Only

% of Time	Function Name
15.99	`SetupFastFullPelSearch`
13.36	`__softboundcets_metadata_load`
8.88	`__softboundcets_allocate_shadow_stack_space`
7.06	`SubPelBlockMotionSearch`
5.35	`__softboundcets_store_base_shadow_stack`
5.14	`__softboundcets_load_base_shadow_stack`
4.98	`__softboundcets_deallocate_shadow_stack_space`
4.27	`__softboundcets_load_bound_shadow_stack`
4.05	`FastPelY_14`
3.82	`__softboundcets_store_bound_shadow_stack`
2.97	`__softboundcets_metadata_store`
2.95	`FastFullPelBlockMotionSearch`
2.74	`FastLine16Y_11`
2.19	`SATD`
2.08	`UMVLine16Y_11`

5 Related Work

Rinard et al. [Rinard 2004] implemented an interesting extension to the Ruwase and Lam object-table-based memory safety mechanism [Ruwase 2004]. Whenever a store is attempted through an out-of-bounds pointer, they store the value in a hash table indexed by the intended referent and the offending offset. If a read is later attempted at the same offset from the object, the value from the hash table is returned. This has the effect of increasing the size of objects as necessary to accommodate all accesses to them. It introduces the possibility of a denial-of-service attack by potentially filling up memory, but in return it not only eliminates buffer overflows but also allows the program to continue functioning as intended rather than aborting upon an out-of-bounds access.

After this project concluded, Intel announced a set of future processor extensions called the Memory Protection Extensions (MPX) [Intel 2013]. They are based on SoftBound's pointer-table-based bookkeeping method. MPX includes new bounds registers, along with new instructions to accelerate both metadata handling and bounds checking. Because the bookkeeping overhead is addressed in addition to bounds checks, there may be hope for future bounds checking that is fast enough for production mode in performance-sensitive legacy applications.

For new code or sufficiently easy-to-convert legacy codebases, Ironclad C++ combines type-safe language subsetting with bounds checks where necessary for impressive reductions in overhead [DeLozier 2013].

6 Future Work

At this time, SAFECode and SoftBound are most suitable for single-threaded applications. They may work in the presence of multithreading, but generally introduce race conditions that can cause incorrect checking. A bounds checking system that tracks intended referents and works in multithreading environments is a worthwhile undertaking.

Another valuable study would be to explore performance gains on future Intel processors that include MPX hardware assistance. If performance proves adequate, then buffer overflow checking could be left in place during production mode. In that case, it might be useful to adapt the technique of Rinard et al. to the pointer-table-based mechanism so that production mode applications could continue running correctly even after out-of-bounds accesses.

We hoisted bounds checks across function boundaries at compile time but not at link time because of difficulties integrating with LLVM's link-time optimization mechanism. It would be interesting to see if performing that optimization at link time would create a significant benefit, given that it would be balanced against link-time inlining performing much of the work at that stage.

7 Conclusion

Buffer overflow is still the most serious problem in software security. The C language is prone to buffer overflows, but it provides benefits too compelling for some application areas to abandon it, and the installed base of legacy C code is enormous. Mechanisms are therefore needed to provide memory safety checking by tracking intended referents. For maximum benefit to legacy code, such mechanisms should not require changes to source code or ABIs and should be able to check some code while linking with unchecked binary libraries.

SAFECode and SoftBound both meet these criteria. It is useful to have an independent test of their performance showing what is achievable by a user downloading the source code. We find that this type of memory safety checking is expensive. We were able to improve performance by hoisting bounds checks out of loops and checking only stores and not loads, and the result is useful for many situations. However, the performance is still inadequate for use in production mode on performance-sensitive applications. SAFECode would benefit most from work on its checking overhead, and SoftBound would benefit most from work on its bookkeeping overhead, especially shadow stack manipulation.

If Intel's proposed new Memory Protection Extensions make it into actual hardware, there is hope for substantial improvement in performance.

References/Bibliography

URLs are valid as of the publication date of this document.

[Criswell 2011]
Criswell, John. *Huge overhead as a result of checks being not inlined* [e-mail sent to the SVADEV mailing list], [online]. Available email: *svadev@cs.uiuc.edu* (December 2011). http://lists.cs.uiuc.edu/pipermail/svadev/2011-December/000167.html

[DeLozier 2013]
DeLozier, Christian, Eisenberg, Richard A., Nagarakatte, Santosh, Osera, Peter-Michael, Martin, Milo, & Zdancewic, Stephan A. *Ironclad C++: A Library-Augmented Type-Safe Subset of C++* (MS-CIS-13-05). University of Pennsylvania CIS, 2013. http://repository.upenn.edu/cis_reports/982/

[Dhurjati 2006a]
Dhurjati, Dinakar & Adve, Vikram. "Backwards-Compatible Array Bounds Checking for C with Very Low Overhead," 162–171. *International Conference on Software Engineering 2006, ICSE 06'*. Shanghai, China, May 2006. IEEE Computer Society, 2006.

[Dhurjati 2006b]
Dhurjati, Dinakar, Kowshik, Sumant & Adve, Vikram. "SAFECode: Enforcing Alias Analysis for Weakly Typed Languages," 144–157. *PLDI 2006 - Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Ottawa, ON, Canada, June 2006. ACM, 2006.

[Eigler 2003]
Eigler, Frank. "Mudflap: Pointer Use Checking for C/C++," 57–70. *Proceedings of the GCC Developers' Summit 2006.* Ottawa, ON, Canada, June 2006. GNU Compiler Collection (GCC), 2006. https://gcc.gnu.org/wiki/HomePage?action=AttachFile&do=get&target=2003-GCC-Summit-Proceedings.pdf

[Intel 2013]
Intel, RB. *Introduction to Intel® Memory Protection Extensions.* Intel, 2013. http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions

[Jones 1997]
Jones, Richard W. M. & Kelly, Paul H. J. "Backwards Compatible Bounds Checking for Arrays and Pointers in C Programs," 13–26. *AADEBUG 97, Proceedings of the 3rd International Workshop of Automatic Debugging.* Linkoping, Sweden, May 1997. Linkoping University Electronic Press, 2007. http://www.ep.liu.se/ea/cis/1997/009/02/index.html

[Nagarakatte 2009]
Nagarakatte, Santosh, Zhao, Jianzhou, Martin, Milo M. K., & Zdancewic, Steve. "SoftBound: Highly Compatible and Complete Spatial Memory Safety for C," 245–258. *PLDI'09 - Proceed-*

ings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). Dublin, Ireland, June 2009. ACM, 2009.

[Plum 2005]
Plum, Thomas & Keaton, David. "Eliminating Buffer Overflows, Using the Compiler or a Standalone Tool," 75–81. *Proceedings of Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM '05): co-located with the Automated Software Engineering Conference 2005 (ASE '05)*. Long Beach, CA, November 2005. NIST, 2006. http://hissa.nist.gov/~black/Papers/NIST%20SP%20500-265.pdf

[Rinard 2004]
Rinard, Martin, Cadar, Cristian, Dumitran, Daniel, Roy, Daniel M., & Leu, Tudor. "A Dynamic Technique for Eliminating Buffer Overflow Vulnerabilities (and Other Memory Errors)," (82–90). *Proceedings - 20th Annual Computer Security Applications Conference (ACSAC*). Tucson, AZ, December 2004. IEEE Computer Society, 2004.

[Ruwase 2004]
Ruwase, Olatunji & Lam, Monica. "A Practical Dynamic Buffer Overflow Detector" *Network and Distributed System Security Symposium NDSS '04* (CD ROM). San Diego, CA, February 2004. Internet Society, 2004. http://www.isoc.org/isoc/conferences/ndss/04/proceedings/

[Seacord 2014]
Seacord, Robert. *The CERT® C Coding Standard: 98 Rules for Developing Safe, Reliable, and Secure Systems*, 2nd ed. Addison-Wesley Professional, 2014.

[Serebryany 2012]
Serebryany, Konstantin, Bruening, Derek, Potapenko, Alexander & Vyukov, Dmitriy. "AddressSanitizer: A Fast Address Sanity Checker," 309–318. *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA, June 2012. University of Trier and Schloss Dagstuhl-Leibniz Center for Informatics, 2012. https://www.usenix.org/system/files/tech-schedule/atc12_proceedings_0.pdf

[Younan 2013]
Younan, Yves. "25 Years of Vulnerabilities: 1988–2012." Sourcefire Vulnerability Research Team, 2013. http://vrt-blog.snort.org/2013/03/25-years-of-vulnerabilities-1988-2012.html

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2014	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Performance of Compiler-Assisted Memory Safety Checking	FA8721-05-C-0003

6. AUTHOR(S)

David Keaton & Robert C. Seacord

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2014-TN-014

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Buffer overflows affect a large installed base of C code. This technical note describes the criteria for deploying a compiler-based memory safety checking tool and the performance that can be achieved with two such tools whose source code is freely available. The note then describes a modification to the LLVM compiler to enable hoisting bounds checks from loops and functions. This proof-of-concept prototype has been used to demonstrate how these optimizations can be performed reliably on bounds checks to improve their performance. However, the performance of bounds propagation is the dominant cost, and the overall runtime cost for bounds checking for C remains expensive, even after these optimizations are applied. Nevertheless, optimized bounds checks are adequate for non-performance-critical applications, and improvements in processor technology may allow optimized bounds checking to be used with performance-critical applications.

14. SUBJECT TERMS	15. NUMBER OF PAGES
buffer overflow; bounds checking	31

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL