Light-weight Bounds Checking[*]

Niranjan Hasabnis, Ashish Misra and R. Sekar
Stony Brook University, Stony Brook, NY 11794

ABSTRACT

Memory errors in C and C++ programs continue to be one of the dominant sources of security problems, accounting for over a third of the high severity vulnerabilities reported in 2011. Wide-spread deployment of defenses such as address-space layout randomization (ASLR) have made memory exploit development more difficult, but recent trends indicate that attacks are evolving to overcome this defense. Techniques for systematic detection and blocking of memory errors can provide more comprehensive protection that can stand up to skilled adversaries, but unfortunately, these techniques introduce much higher overheads and provide significantly less compatibility than ASLR. We propose a new memory error detection technique that explores a part of the design space that trades off some ability to detect bounds errors in order to obtain good performance and excellent backwards compatibility. On the SPECINT 2000 benchmark, the runtime overheads of our technique is about half of that reported by the fastest previous bounds-checking technique. On the compatibility front, our technique has been tested on over 7 million lines of code, which is much larger than that reported for previous bounds-checking techniques.

1. Introduction

Memory errors in C and C++ programs constitute one of the most difficult class of errors to track down and debug. They remain an important source of problems long after programs are tested and shipped. From a security perspective, memory corruption vulnerabilities continue to be a dominant concern: they account for four of the CWE/SANS "Top 25 Most Dangerous Software Errors" [35]. According to the NVD database [27] from NIST, of the 1818 high severity vulnerabilities reported in 2011, over one-third were due to memory corruption. They are also behind most of critical updates distributed by OS and application vendors.

Recognizing the importance of memory errors, numerous research efforts, spanning over a quarter century, have targeted runtime detection of memory errors [18, 34, 15, 6, 29, 17, 16, 25, 32, 38, 12, 13, 26, 5, 22, 39, 3, 23]. These efforts have yielded many practical tools that enjoy widespread use during software testing. However, in order to defend against security exploits, these techniques have to be deployed in operational software. Two factors have so far held this up:

- *High runtime overheads.* Many of these techniques suffer from significant runtime overheads, often reaching or exceeding 100% for compute-intensive programs.
- *Incompatibility with existing code.* Some of these techniques are not compatible with precompiled libraries. In addition, they are incompatible with one or more of the following features found in some C/C++ programs: use of address arithmetic and type-casting, conversion between integer and pointer types, and bit-manipulations on pointers such as packing a pointer with additional data into a single word[1], or the use of bit-masking to force a pointer to be in-bounds.

For these reasons, most contemporary operating systems (including Microsoft Windows and Linux) have based their memory corruption vulnerability defenses on address-space layout randomization (ASLR) [8, 30] and data execution prevention (DEP). Rather than being concerned with program operations involving pointers, these techniques shift their focus to the after-effects of memory corruption. In particular, both absolute-address randomization [8, 30] and relative address randomization [10, 19] do not interfere with a program's ability to manipulate pointers, thus avoiding compatibility issues; instead, they randomize the effect of dereferencing an out-of-bounds pointer. Since ASLR does not check any pointer operation, it poses low or negligible runtime overhead. DEP is not concerned with memory corruption at all, but one of its common after-effects, namely, execution of injected code, which is prevented by ensuring that no (injected) data can be executed.

Widespread deployment of ASLR and DEP has raised the bar for successful exploits. However, it appears that this may be a temporary rather than permanent setback for attackers. Over the past few years, *heapspray* attacks [2] have become a standard technique to brute-force ASLR. Even when deployed together with DEP, attackers have been able to bypass ASLR, as demonstrated in a recent high-profile zero-day exploit on Adobe Acrobat/Reader [24][2].

[*]This work was supported in part by an NSF grant CNS-0831298, an AFOSR grant FA9550-09-1-0539, and an ONR grant N000140710928.

[1]This may allow reuse of bits whose values are known, e.g., lower 3 bits of a 8-byte aligned pointer.

[2]To overcome DEP, these exploits use return-to-libc attacks, or return-oriented programming [33, 11].

```
char *beg, *end;          driver() {
int get(unsigned i) {       // makes many calls to get()
  if ((beg+i) < end)        beg = (char*) malloc(1024);
    return *(beg+i);        end = beg+1024;
  else return -1;           for (int i=0; i < N; i++)
}                             get(i);
                          }
```

Figure 1: Program causing build-up of OOB objects

```
char p[10];
char *q = &p[30]; // creates an OOB pointer
long r = (long)q;
*((char *)(r-23)) = 'x'; // Assigns to p[7];
```

Figure 2: Casting between OOB pointer and integer

```
char *p, *q;
...
ptrdiff_t o = p-q;
*(q+o) = 'x'; // Has the same effect as *p = 'x'
```

Figure 3: Using the difference between two pointers

The weakness of ASLR is that it is a *probabilistic* defense. Moreover, neither ASLR nor DEP prevent memory errors, but an after-effect that may occur long afterwards. This interval provides an opportunity for attackers to devise clever attacks that bypass these defenses.

To defend reliably against adversarial attacks, it is therefore necessary to bring back the strengths of memory error detection techniques, namely, *prompt* and *deterministic* identification of memory corruptions. At the same time, in order to enable widespread deployment, it is necessary to provide the same level of compatibility that ASLR/DEP provide. We believe this can only be achieved if we follow the design philosophy of ASLR: *avoid any restrictions to pointer manipulation operations, and shift the focus to preventing invalid pointer dereferences.* We develop a new memory error detection technique that achieves this combination of strengths, and thus provides a better basis for long-lasting defense against clever attacks by skilled adversaries. Below, we provide an overview of our approach and summarize its contributions.

1.1 Background and Rationale

Memory errors are broadly divided into *spatial errors* and *temporal errors*. Our focus is on spatial errors, which contribute to most of memory-related security vulnerabilities.

Many existing spatial error detection techniques [6, 29, 25, 38, 26, 22] maintain per-pointer metadata, e.g., the base address and size of the referent (i.e., the memory region pointed). This data can be used to detect instances when a pointer value goes outside of its referent, either due to pointer arithmetic or due to an unintended overwrite of the pointer[3]. Unfortunately, such techniques are incompatible with uninstrumented libraries, i.e., precompiled libraries that do not incorporate memory error checking. This is because the required per-pointer metadata would be absent for data structures created within such libraries. This factor was cited as the motivation for the bounds-checking technique of Jones and Kelly [17], which has been further refined by many researchers [32, 12, 5, 39].

[3]Such an overwrite is possible due to features such as arbitrary casting and unions. For instance, a structure containing pointers may be cast into a byte array, and arbitrary values copied into it.

Bounds-checking techniques maintain metadata regarding allocations, but no per-pointer metadata is maintained. Each variable on the stack or on the static area corresponds to a distinct allocation unit, as does a block returned by `malloc`. The property enforced by these techniques is that if a pointer p is derived from q using pointer arithmetic, then both p and q should fall within the bounds of the same allocation unit. If not, p is assigned a special value `ILLEGAL` [17] that points to inaccessible memory. Thus, any future attempt to dereference p would lead to a memory fault.

Although Jones and Kelly's technique [17] was designed to be compatible with the ANSI C standard, Ruwase and Lam [32] noted that a large number of existing programs (about 60% among the ones they studied) violate this standard and are hence broken by Jones and Kelly's technique. In particular, they create an out-of-bounds pointer p, but use pointer arithmetic to bring it back within bounds. Unfortunately, the original pointer value would have been lost when p was assigned the value `ILLEGAL`. CRED [32] incorporates a clever approach to overcome this problem: p is set to point to an *out-of-bounds* (OOB) object, which in turn stores the (out-of-bounds) value of p, as well as the original referent of p. CRED was shown to work on many medium to large software packages. Subsequent bounds-checking works [12, 5, 39] have significantly improved on the performance of CRED, but have not focused on the compatibility problems posed by these OOB objects. Unfortunately, OOB objects do pose several challenges in practice:

- *Unpredictable space usage for storing OOB objects.* Figure 1 shows a program that uses about 1KB of data memory and contains no memory errors. With CRED[4], each iteration of the loop with `i` $>$ 1024 causes the creation of one OOB object. If the loop is executed 500K times, the program consumes 7MB rather than 1KB! With more iterations, it will eventually run out of memory.
- *High runtime overheads.* Every pointer arithmetic and dereference operation needs additional checks to determine if OOB objects are involved, thus further increasing overheads. For instance, 500K iterations of the loop in Figure 1 takes a small fraction of a second on a contemporary laptop, but this increases to 15 minutes with CRED due to OOB operations.

  Although Reference [12] makes some changes to CRED's OOB object design, these only improve the efficiency of dereference operations, and do not change aspects of OOB design that are related to these problems. As a result References [12, 39] continue to suffer from unpredictable space and runtime overheads for OOB objects. Baggy [5] does use a different scheme for OOB objects, but this scheme restricts the usable range of OOB pointers to half the size of the referent object. This restriction seems to be the most significant compatibility issue posed by Baggy. It caused Baggy to fail on some SPEC 2000 benchmarks, requiring a few manual changes to these programs[5].
- *Potential to break working programs.* Since out-of-bounds pointers have a different representation from valid pointers, they can cause compatibility problems if they are passed to uninstrumented libraries. Two other examples that are broken due to the use of OOB are shown in Fig-

[4]We used CRED's Bounds Checking Patches for gcc-4.0.2.
[5]As reported in the evaluation section of their work, they had to modify `perlbmk` and `gap` of SPEC 2000 benchmark.

ures 2 and 3. Figure 3 captures a use found in the SPEC 2000's `gcc` program that caused Baggy to fail.

1.2 Approach Overview and Contributions

For reasons enumerated above, we argue that achieving full backward compatibility with existing software requires techniques that neither use per-pointer metadata, nor change the representation of pointers under any condition. This seems to leave us with only one option, namely, changing the memory layout in a manner that allows us to detect (most) memory errors. In particular, we base our Lightweight Bounds Checking (LBC) on the idea of *guard zones* around memory objects. Dereferencing a pointer to the guard zone will flag a memory error. The vast majority of bounds errors involve accesses that are just past the end of an object, and these can be detected using guard zones.

Although LBC is applicable to C and C++ programs, our current implementation, available at http://seclab.cs.sunysb.edu/download.html, is limited to C-programs.

The idea of guard zones is quite old, and is used in tools such as Purify [15]. What is new here is the development of techniques for implementing them *efficiently.* In particular, this paper makes the following contributions:

- *Low memory overheads.* The fixed overheads (i.e., overhead that is independent of the memory usage of a program) of our approach is just 256KB. Its variable overhead for the SPEC 2000 benchmark suite is less than 5%, which compares favorably with Baggy's memory overhead of about 15% on the same suite.
- *Low runtime overheads.* On the SPECINT 2000 benchmark, our CPU overhead of 23% is less than half of that reported by Baggy [5], the fastest bounds checking implementation available. This high performance is achieved using the following techniques:
  - *Zero metadata operations in the common case.* Memory error detection techniques introduce additional memory accesses to read or write metadata, thus increasing the pressure on caches and memory. We develop a design that improves performance by avoiding additional metadata operations for most memory reads.
  - *Avoidance of locking for most metadata operations.* Metadata reads and updates occur very frequently in LBC. Use of locks to guard these operations can lead to high overheads. We therefore develop new techniques to minimize the number of lock operations, leading to overhead reduction by almost a factor of two.
  - *Static analysis to eliminate metadata.* We develop a static analysis based on CCured's [25] pointer analysis. Overheads are reduced another two-fold as a result.
- *Highly backwards compatible.* The only programs that should break under LBC are those that make assumptions regarding the relative distances between different variables. Moreover, LBC does not interfere with a program's ability to allocate large amounts of memory. Unlike Baggy, which relies on a buddy allocator, our approach is compatible with the more traditional chunk allocators. To demonstrate compatibility, we have successfully compiled and tested over 7M lines of C-code, which is about 7 times more than any of the previous techniques.
- *Effective in blocking exploits.* Although LBC isn't guaranteed to detect all memory errors, our evaluation on the bugbench [20], SAMATE [28] and the RIPE [36] benchmarks show its effectiveness for detecting buffer overflows and blocking security exploits.

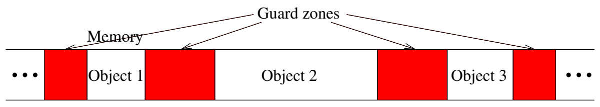

Figure 4: Illustration of guard zones

2. LBC Overview

LBC separates all objects[6] in all memory areas — the stack, heap and static area — from each other using guard zones as shown in Figure 4. Guard zones represent memory that must not be accessed by any correct program.

The semantics of many pointer manipulation operations are left unspecified in the C-language. However, most compilers permit free, value-preserving conversions between pointers and integers. This has led to a belief held by many C programmers that pointers and integers (or long integers) are interchangeable, and that they may use arbitrary pointer arithmetic. For this reason, LBC leaves pointer arithmetic (and pointer-integer conversion) operations alone; instead, it only checks pointer dereferences. In particular, LBC instruments every memory read and write of `*p` to check if `p` points to the guard zone, and if so, aborts the program with an error message[7]. Note that since errors are detected before memory has been corrupted, recovery is easier with LBC as compared to alternatives such as ASLR.

2.1 Guard zone size

A large size for guard zones increases the likelihood that out-of-bounds pointers will land in the guard zone, and thus increases the likelihood of detecting out-of-bounds dereferences. At the same time, too large a size will increase space overheads. Runtime will increase as well due to increased time needed for initializing guard zones. Making the right trade-off requires some consideration of how a memory object is used. For instance, if the object is an array, the guard zone size should be a small multiple of the array element size: since a guard zone n times the element size will enable the detection of overflow by up to n elements. If the object is not an array, the guard zone may be made a fraction of the allocation request. More generally, we may use the following expression to determine guard zone size.

$$max(2 * element\_size, request\_size/8)$$

Our implementation sets a minimum size of 8 bytes and maximum size of 1K for guard zones.

Note that complete type information is available for stack allocations, and hence we can determine element size. For calls to `malloc` and related routines, note that they typically have the following structure:

$$\texttt{x} = (\texttt{T}*)\texttt{malloc}(\texttt{n} * \texttt{sizeof}(\texttt{T}))$$

Element size in this case is obtained from the pointer type to which malloc result is cast. For others, we can fall back on a default value, say, the word size on the underlying processor.

[6]An object refers to a unit of allocation in C programs: a variable on the stack or global memory, or a block returned by `malloc`.
[7]An alternative is to call a handler specified by the program.

`Original program`	`LBC-transformed version`
`void f() {`	`void f() {`
`  struct X {`	`  struct X {`
`    char *p;`	`    char *p;`
`    int q[8];`	`    int q[8];`
`  } x;`	`  };`
	`  struct X_gz {`
	`    char front[24];`
	`    struct X orig;`
	`    char rear[24];`
	`  } x;`
`  struct X* y;`	`  struct X* y;`
	`  init_guardzone(x);`
	`  init_guardzonemap(x);`
`  y=&x;`	`  y=&x.orig;`
`  *y->p = 'a';`	`  if (y->p==guardzone_val)`
	`    if slowcheck(y->p)`
	`      flag_error();`
	`  if (*y->p==guardzone_val)`
	`    if slowcheck(*y->p)`
	`      flag_error();`
	`  *y->p = 'a';`
	`  uninit_guardzonemap(x);`
`}`	`}`

Figure 5: A sample program and its transformation.

Note that in comparison to stack and heap allocations, the number of allocations in the static area are typically much smaller. Hence we can use larger sizes (say, a minimum of $4 * element\_size$) for guard zones in the static area.

2.2 Guard zone representation and access

Memory error detection techniques are notorious for high overheads even when the underlying checks seem quite simple and efficient. An important reason for this is that they typically increase the number of memory accesses, since they need to fetch metadata related to pointers or their referents. Since memory access times are far longer than CPU cycles on modern processors, even a few additional accesses disproportionately increase runtime overheads.

For the above reason, LBC design is geared to achieve *zero additional memory accesses in the most common case.* In particular, we note that most memory accesses are reads, and LBC avoids extra memory accesses for them. This is achieved as follows. LBC initializes guard zones with a special *guard zone value.* This enables a *fast check* in the typical case, which simply involves determining that the data just read was *not* the guard zone value. If the guard zone value is a random 8-bit value, then a match with guard zone value is likely to happen $1/256^{\text{th}}$ of the time. In this atypical case, a more expensive *slow check* is performed, as described below.

The slow check operation makes use of a *guard map.* This map associates one bit of data for each byte of data memory to specify if that byte falls on a guard zone. A simple and efficient way to implement such a map is to use an array that is indexed by memory address. The downside is that this array requires a contiguous memory region that is as large as $1/8^{\text{th}}$ of the addressable memory. Moreover, to obtain maximum performance, the array may have to be mapped to a fixed memory location [37]. These aspects pose compatibility problems for applications that require large contiguous memory spaces, and applications that need to fix the memory location of certain objects. Hence we opt for a two-level data structure. The first level consists of a static array of n elements, each of which points to a second-level structure of size m-bits. These second-level structures, called *map pages*, are allocated on demand. Note that $n * m$ should equal the size of the virtual address space. Smaller values for n will reduce the fixed memory overheads, i.e., overheads that are independent of the memory usage of a program. Our implementation sets $n = m = 2^{16}$, i.e., it uses a 256KB static array with 8KB map pages to address a total of 2^{32} bytes.

To look up the guard map for a data location A, we first access the static array to identify the map page pointer corresponding to A. If this pointer is non-null, then the bit corresponding to location A is accessed. If the map page pointer is null, then there are two cases to consider. For pointer dereferences, this means that the memory location being looked up is not in the guard zone. On the other hand, if this is an operation to initialize a guard zone, then the map page has to be allocated and initialized.

Note that the guard map needs to store only the locations of guard zones; it is unnecessary to have map pages corresponding to memory regions that have no guard zones. This means that for programs that make large allocations, the guard map can take less than $1/8^{\text{th}}$ of allocated memory.

2.3 Instrumentation strategy

Two main strategies for implementing LBC are: (a) modify an existing compiler such as gcc, and (b) a *source-to-source* transformation. Although the key elements of LBC design are unaffected by this choice, we preferred the latter choice due to its compiler-neutral nature. Key aspects of our source transformation are:

- *Changing the types of static and stack-allocated variables* to introduce guard zones. The new type consists of three fields: the front guard zone, a field with the original type of the variable, and a rear guard zone. Figure 5 illustrates this transformation for the variable `x`. Dynamic memory allocations are handled by a runtime library, and do not require source-level changes.
- *Initialization of guard zones.* Code is emitted at the beginning of each function to initialize the guard zone and guard map corresponding to the local variables of that function. In addition, code is added to the constructor section of each source file to initialize guard zones of global variables.

  Note that when a variable is deallocated, the above steps need to be reversed. However, we do not clear the guard zones, but only the guard map. This approach may seem to increase the likelihood that reads would return a guard zone value, and hence increase the probability of calling a slow check. However, note that this can happen only for uninitialized reads (should not occur unless there is a bug) or the first write after an allocation. Thus, omission of unintialization can be a net positive.
- *Replacement of variable references* so that they refer to the start of the original variables. In Figure 5, original references to `x` are replaced with `x.orig`.
- *Pointer dereferencing instrumentation* to perform guard zone checks, as described earlier.

2.4 Runtime support functions

LBC modifies `malloc` and related routines to allocate additional space needed for front and rear guard zones, and initialize them. Another component of runtime support is a library to implement the guard map data structure.

Our runtime introduces a wrapper for the standard `malloc` function. This wrapper increases the size of memory request by $2 * gz\_size$, where *gz_size* denotes the size of guard zone for the memory block. This new size request is passed on to the original `malloc`. When it returns, the wrapper initializes the front and rear guard zones, and returns a pointer that is *gz_size* from the start of the block.

Wrappers are also created for other dynamic memory allocation functions such as `calloc`, `realloc`, `alloca`, and `memalign`. Variants of these wrappers are also created that take an additional parameter that specifies the element size, in case of requests that allocate space for an array. As noted before, if this element size can be easily inferred from the source code, then the transformation can call these variants, providing them with more information to assist them in choosing the most appropriate guard zone size.

A catch with `memalign` is that the alignment requirements may be large. Since the guard zone size must be a multiple of alignment size, this may cause a large amount of memory to be wasted. It is possible to trade off some protection for reduction in space use: in particular, a front guard zone is omitted if its size must be greater than the size chosen for the rear guard zone.

Calls to `free` are also wrapped. Note that the pointer p returned to this wrapper will be offset by *gz_size* from the beginning of the block that was allocated by `malloc`. Since *gz_size* was computed at malloc-time but not remembered, our approach for computing the beginning of the block uses the guard map. Specifically, we scan backwards in the map, starting from the bit location corresponding to p, to determine where the front guard zone ends.

2.5 Separate compilation and libraries

In order to preserve compatibility with existing code, LBC supports separate compilation of each source file.

LBC provides full compatibility with uninstrumented libraries. Note that any memory dynamically allocated by such libraries will be intercepted by LBC's `malloc` wrappers and guard zones added, similar to memory blocks dynamically allocated by the rest of the program. Stack and static memory allocated by uninstrumented libraries will contain no guard zones. Consequently, dereferences of pointers to stack or static memory objects of uninstrumented libraries will always indicate that they do not point to guard zones. Thus, LBC will not interfere with access to these objects. Naturally, this means that memory errors in the stack and static areas of uninstrumented libraries cannot be detected.

3. Optimizations

3.1 Simple optimizations

There are several simple optimizations employed by LBC:

- *Speeding up guard zone checks for all primitive types.* Our original design treated guard zone value as a single byte. Although the use of a 4-byte guard-zone value for integers would further reduce the ratio of slow checks performed, we felt that the actual gains would be negligible since slow checks were already rare. So, we were surprised to find out that on many integer benchmarks, LBC's overheads were quite high. Further analysis revealed that the compiler generated code was performing two memory reads: one to read a byte value, and another to read an integer value. Our optimization was to match the guard zone value with the size and type of the pointer being dereferenced. Now, we were able to achieve the goal of zero additional memory reads for the fast check operation.
- *Removal of guard zones for "safe" variables.* These are local variables and static global variables (i.e., variables identified with the keyword *static*) that contain no arrays, and their addresses are never taken. As observed in previous works, they cannot be involved in bounds errors.
- *Optimizations such as common subexpression elimination.* Programs often contain repeated pointer dereferences, e.g.,

  ```
  p->q->r = p->q->r + p->q->s
  ```

  For good performance, we need to eliminate redundant pointer checks. Since our approach relies on a source-to-source transformation, we leave it to the C-compiler (used to compile the transformed code) to perform these optimizations. The only effort needed in our implementation is to ensure that our transformation does not obscure possible optimizations, or make them difficult.
- *Guard zone placement.* Note that the design described so far incorporates guard zones on both sides of each object. By sharing the guard zone between two successive objects, we can reduce both time and space overheads. This is accomplished as follows:
  - *Stack objects:* To control the lay out of stack allocated objects, our transformation creates a single struct that includes all of the unsafe local variables. A guard zone is introduced at the front and rear of this struct, and in between successive fields that each represent a local variable in the original program. The size of each guard zone is chosen to be the maximum of the sizes needed for protecting the variables that are adjacent to it. To minimize the size of these guard zones, our transformation lists the original variables in increasing order of their required guard zone sizes.
  - *Heap objects:* Note that for heap-allocated objects, typical heap implementations place heap metadata just before user data. The front guard zone protects this metadata, while the rear guard zone protects the metadata of an adjacent heap block. For this reason, we do leave both guard zones in place for the heap.
  - *Global objects:* As mentioned earlier, global objects are relatively few in number and are initialized just once, so we did not consider additional optimizations for them.

3.2 Multithreaded programs

In a multithreaded program, LBC introduces the potential for new race conditions. Note that each pointer dereference is now preceded by metadata accesses, i.e., the guard zones and the guard map. While one thread is reading this metadata, another thread may concurrently update it.

The safest approach for avoiding these race conditions is to use locks before accessing guard zone or guard map. However, such an approach leads to major overheads in LBC. We have therefore developed an approach to minimize the use of locks while still ensuring correctness, by considering all pairs of conflicting operations.

- *Multiple concurrent dereferencing operations.* Dereferencing operations perform only reads on guard zones and guard map. Thus, multiple read operations can be executed in parallel without experiencing concurrency errors.

- *Concurrent initializations of guard zones.* Initialization of a guard zone follows the allocation of an object. A correct compiler and runtime infrastructure must already ensure that no two concurrent allocations will overlap. Note that except for stack-allocated objects, a guard zone is associated with exactly one object. Thus, the guard zones being concurrently initialized must also be disjoint, thereby avoiding race conditions. Stack objects are allocated by a single thread, thus precluding concurrency errors.
- *Guard zone initialization concurrent with the dereferencing of a pointer into that guard zone.* Note that LBC seeks to detect those memory errors where arithmetic on a pointer to an object caused it to point to the adjacent guard zone. LBC, in general, cannot assure the absence of other types of memory errors such as unintended overwriting of pointers or overflows that jump past guard zones. Thus, the only case in which we need to eliminate the possibility of a concurrency error is one where the pointer being dereferenced is derived from a pointer to the object being allocated (which in turn caused guard zone initialization). However, this cannot happen: a pointer to the object is invalid until the object is initialized.
- *Concurrent initializations of guard map.* These should occur in response to two concurrent object allocations.
  - If the two initializations fall on different map pages, then there is obviously no conflict.
  - If they fall on the same map page that is already present, we can again rule out the possibility of conflict because the regions updated for disjoint object allocations must also be disjoint. However, since individual bits cannot be atomically updated on most architectures, it is possible that the updates overlap even though the bit positions that need updating are disjoint. Note that this danger arises for concurrent allocations of objects that are next to each other. This cannot happen on the stack since allocations on a single stack happen sequentially. It cannot happen on the heap on most 32-bit architectures because malloc alignment requirements ensure that guard zones for heap variables are aligned on an 8-byte boundary, and hence the corresponding bits in the guard map must be aligned on a byte boundary. In the static area, we explicitly ensure an alignment of guard zones on 8-byte boundaries to avoid this problem.
  - If they fall on the same map page that is not present, then LBC uses a lock. In particular, if the first level look up yields a null pointer, then a lock is acquired. After acquiring the lock, the pointer is checked again, and if it is still empty, a new map page is allocated and the pointer updated. Finally, the lock is released, and the map page bits are updated as appropriate.
- *Guard map initialization concurrent with pointer dereferencing.* As noted before, these two concurrent accesses must involve different objects. Also, as argued in the previous case, conflicts cannot arise due to access to guard zone bits, but only when the relevant map page is not present. However, in this case, it must be the case that the pointer being dereferenced does not correspond to a guard zone, or else the map page would have been created at the allocation time of the object being dereferenced. Thus, pointer dereferencing operations simply return if a map page pointer is null, and does not use locks.

Our optimization avoids locks except in the rare case of a map page allocation. Note that each time the lock is acquired, a map page is added. Since map pages are never deleted, our design has the interesting property that the total number of lock operations is bounded, and is a small fraction of the address space used by the application.

3.3 Static Analysis

There may be many pointer variables in a program that are never involved in pointer arithmetic. Clearly, guard zone checks do not provide any additional protection for such pointer variables. We have developed a static analysis to detect such pointers. Our analysis mainly uses the static analysis implemented in CCured [25]. In particular, they identify three kinds of pointers. *Safe* pointers are never involved in pointer arithmetic or unsafe casts[8]. *Sequential* pointers are those that are involved in pointer arithmetic but not unsafe casts. Finally, *wild* pointers may be involved in both pointer arithmetic and unsafe casts.

A type inference algorithm for inferring pointer types is described in [25] and is implemented in the CCured prototype. We made a few changes to this algorithm. In particular, pointers inferred to be safe are exactly those that do not need guard zone checks. Pointers inferred to be sequential, as well as those inferred to be wild, will be instrumented for guard zone checks.

There is a small but important change we made that to the type system. In particular, CCured permits a sequential pointer q to be assigned to a safe pointer p. To make this sound, CCured introduces a bounds check before assigning to p, and if the check fails, program execution is aborted. However, this may be premature, and break legitimate programs. In particular, if the program never dereferences p, then it will never be involved in a memory error, yet CCured would have aborted the program. To avoid this possibility, we have modified the type inference algorithm to disallow casts from sequential to safe pointers.

Note that objects whose addresses are assigned only to safe pointers (or none at all) do not need any guard zones.

Static analysis becomes a bit complex in the context of separate compilation. In particular, if a function f in one C-file calls another function g in another C-file, then we are unable to analyze how g uses the parameters passed to it by f. In our current implementation, we label any pointer variable passed to such an external function and all global pointers as a wild pointer. This is sound, but can obviously impact precision, as wild pointers tend to propagate rapidly during type inference. We are currently investigating ways to improve the precision further in this case.

To overcome the challenge of precision loss posed by separate compilation, CIL [21] provides a "merged compilation" mode that works with most makefiles, and merges all relevant C-files into one big file that is then analyzed. This enables inter-procedural analysis across procedures defined in difference source files. Unfortunately, this merged mode compilation can fail for some large packages. Our current approach is to invoke a merge-mode compilation for any pack-

[8]Intuitively, unsafe casts are those that can cause "confusion" about the nature of pointers contained within a referent, e.g., casting an `int` to a `char *`, or a `struct A*` to a `struct B*` where `A` contains no pointers but `B` does.

age, but if the package encounters a build failure, then it is rebuilt using separate compilation. Many of the larger packages in our evaluation relied on separate compilation, but LBC still provided good performance.

Unlike some approaches that *require* global compilation, our use of merged mode compilation is opportunistic: it is used if it works. LBC does not require access to all source code, but benefits from source code that is available. For large projects that organize their source code into modules that are built independently, LBC can benefit from interprocedural analysis across source files within a module, while permitting separate compilation of each module. Most importantly, our opportunistic use of merged mode compilation is fully compatible with external libraries (including shared libraries) that are available only in binary form.

Certain additional optimizations are possible for standard libraries. For libraries where CCured provides accurate type information, e.g., `glibc`, LBC uses this information instead of treating pointer parameters to these libraries as wild. For other libraries, we make use of `const` annotations to eliminate designation of some pointers as wild. In particular, an otherwise safe pointer can be passed to an external library in the place of a `const` pointer.

4. Experimental Evaluation

In this section, we evaluate LBC in terms of (a) compatibility with existing software, (b) runtime and space overhead, (c) effectiveness of various optimizations, and (d) ability to detect memory errors and stop exploits.

4.1 Implementation

We implemented LBC for 32-bit machines running Linux. Our prototype consists of a program analysis/transformation system implemented using CIL [21], together with C-code that implements runtime support functions. CIL infrastructure provides a high-level tree representation of a C-program, along with a set of tools that simplify analysis and instrumentation. Our implementation uses two passes, with the first pass determining which objects need guard zones, and the second pass introducing these guard zones and associated instrumentation. Overall, LBC consists of about 5K lines of OCaml code and 1K lines of C code.

4.2 Compatibility

To verify the compatibility of our technique with real world programs, we compiled a variety of software packages shown in Figure 6. All of them compiled successfully. Although it is not possible to comprehensively test each of them, we made sure that they could all run. Where test cases were available with the package, we were able to verify that they produced the expected results.

Our compatibility tests involved 7 million lines of C-code (as measured using SLOCCount tool [1]). This is much larger than those considered in previous works: Baggy [5] and CRED [32] were tested with about 1M lines of code, while SoftBound [22] was tested with 272K lines.

4.3 Runtime Overhead

We measured space and runtime overhead of our technique using CPU-intensive SPEC 2000 benchmark. We used SPEC 2000 rather than more recent ones such as SPEC 2006 because previous works have mostly used SPEC 2000, thus permitting a direct performance comparison. All the benchmarks were compiled with standard options (which includes -O2 optimization) and run using the tests provided with them. The measurements were taken on a Ubuntu 9.04 system with 2.4GHz Intel Core 2 Duo and 3 GB of RAM.

Program	**Description**	**KLOC**
wireshark-1.0.7	Network Traffic Analyzer	1473
binutils-2.13.2.1	Binary Tools	947
emacs-22-22.2	The GNU Emacs Editor	834
ghostscript-9.00	PostScript/PDF Interpreter	714
gimp-2.7.1	Image Manipulation	675
evolution-2.26.1	Email/Personal Organizer	305
pidgin-2.5.5	Multi-protocol IM	296
pine-4.64	Email Client	238
openssl-0.9.8k	SSL/Crypto Suite	235
httpd-2.2.16	Apache Web Server	230
snort-2.7.0	Intrusion Detection	115
gnupg-1.4.11	OpenPGP Implementation	115
sendmail-8.14.4	Email Server	92
postfix-2.6.7	Email Server	90
coreutils-8.9	UNIX Utilities	87
libcurl-7.21.3	File Transfer Suite	82
xpdf-3.02	PDF Viewer	82
openssh-4.7	SSH Server/Client	60
transmission-1.51	BitTorrent Client	59
audacious-1.5.1	Audio Player	56
squid-3.1.0.9	Caching Web Proxy	37
hypermail-2.3.0	HTML Converter	36
gawk-3.1.8	String Manipulation Tool	32
bison-2.4.3	Parser Generator	27
libpng-1.4.1	PNG Library	25
enscript-1.6.1	ASCII to Postscript converter	22
wu-ftpd-2.8.0	FTP Server	22
ccrypt-1.9	Encryption Utility	13
grep-2.7	Pattern Matching Utility	11
monkey-0.12.2	Web Server	10
WsMp3-0.0.10	Web Server	4
pgp4pine-1.76	Mail Encryption Tool	3
nullhttpd-0.5.1	Lightweight HTTP Server	2
zlib-1.2.3	Compression Library	0.4
polymorph-0.40	Filesystem Unixier	0.4
Total		**7000**

Figure 6: Packages compiled and run with LBC

Of the SPECINT 2000 benchmark, we left out `eon` which is a C++ program, and `vortex` which failed to compile with the version of gcc we had. We could compile the rest successfully. Note that Baggy [5] could not compile `gcc`, and had to make changes to `perlbmk` and `gap` — all due to out-of-bounds pointer problem discussed in the introduction.

As shown in Figure 7, the average runtime overhead of LBC for SPECINT 2000 benchmarks is 23%. Baggy [5], which has the best performance among previous bounds checkers, reports 2.5 times this overhead — an average of 60%. Baggy's highest overhead of 127% is for `vpr`, for which LBC has only 27% overhead. Of course, it is not a simple numbers comparison since Baggy can detect more memory errors than LBC. However, we believe that LBC's chosen trade-off between performance and compatibility versus coverage is more appropriate when applying bounds checking to large software collections, e.g., GNU/Linux distributions.

On the three C programs that were included in SPECFP 2000, LBC introduced an average overhead of about 9%.

We also measured overheads for a few security-relevant applications: openssl-0.9.8k, apache-2.2.16, nullhttpd-0.5.1,

Program	*Base runtime* (seconds)	*Runtime overhead (%)*	*Base Memory* (in MB)	*Memory overhead (%)*
gzip	156	1	180	0.2
vpr	122	27	20	1.5
gcc	65.6	43	83	7.5
mcf	115	0	94	0.3
crafty	77.7	2	1	44
parser	179	15	30	1.1
perlbmk	109	25	45	0.8
gap	74	54	192	0.2
bzip2	129	21	184	0.2
twolf	174	46	1	30
Average		**23**		**8.5**

Figure 7: SPECINT 2000 performance overheads

and libpng-1.4.1. LBC's CPU overheads were low across all these programs, ranging between 1% and 13%. In particular, LBC's overhead was 12.7% and 3.5% on openssl and libpng respectively, when they were run with the default tests that came with these packages. Apache and Nullhttpd performance was obtained using `ab`, Apache's web server benchmarking tool. Both the server and the client were run on the same machine in order to avoid network bandwidth saturation effects. We measured a decrease in throughput of 7% for apache and 1.5% for Nullhttpd. Baggy's numbers are close to ours. This is partly because the effectiveness of our static analysis was limited due to separate compilation.

4.4 Memory Overhead

Figure 7 also shows memory overheads of LBC. As noted earlier, LBC can be highly space-efficient for large allocations. This is because the guard zone size is capped at 1KB, and because guard map size increases in proportion to the total size of guard zones.

The memory overhead of LBC is 8.5% on SPECINT 2000, which is significantly lower than that of Baggy. `crafty` and `twolf` show relatively higher overhead when compared with other programs. This is primarily because of the fixed overhead (mentioned in the introduction section) which proves to be significant for these programs.

4.5 Effectiveness of Optimizations

Figure 8 summarizes the effectiveness of various LBC optimizations. In this figure, the second column shows the base runtime of the benchmark when compiled using gcc, and without any use of LBC. The next columns show overheads (measured as a percentage of base runtime) when different optimizations were enabled. The fourth column shows the effectiveness of fast check optimization, which reduces overheads from a massive 556% to (a still considerable) 86% on the average. The lock optimization is also very effective, reducing the overhead by another factor of two. Finally, static analysis achieves almost another halving of the overhead.

4.6 Detection of bounds violations

We evaluated LBC on three benchmarks and the Nullhttpd server. As detailed below, LBC detected all bounds violations in all these cases. Note that intra-object overflows and temporal errors are outside the scope of LBC as well as other bounds-checkers [17, 32, 5].

Bugbench [20]. This benchmark consists of several buggy real-world programs, including `bc-1.06`, `cvs-1.11.4`, `gzip-1.2.4`, `htpd1`, `man-1.5h1`, `mysql1`, `mysql2`, `mysql3`, `ncompress`, `squid-2.3`, and `polymorph-0.4.0`. Of these programs, `mysql1`, `mysql2`, and `mysql3` have data race bug, and `cvs` has a double-free bug, both of which are outside the scope of LBC. Of the remaining programs, `bc`, `man`, and `polymorph` have overflow bugs, and LBC could detect them. `gzip` and `ncompress` exploit overflows using `strcpy`, and are detected with an LBC-instrumented version of `glibc`[9].

Program	*Base* (secs)	*No optimization overhead (%)*	*Prev col + Fast check optimization (%)*	*Prev column + Lock optimization (%)*	*Prev column + static analysis (%)*
gzip	156	302	12	4	1
vpr	122	565	81	32	27
gcc	65.6	641	147	55	43
mcf	115	191	9	2	0
crafty	77.7	182	57	48	2
parser	179	465	52	17	15
perlbmk	109	834	152	107	25
gap	74	1234	149	62	54
bzip2	129	599	74	36	21
twolf	174	543	129	49	46
Avg	**120**	**556**	**86**	**41**	**23**

Figure 8: Effectiveness of LBC optimizations

Weakness		**SRD test case ID**
CWE-121	Stack overflow	9, 14, 115, 601, 751, 907
CWE-122	Heap overflow	015, 145, 147, 572
CWE-123	Write-where-what	013, 750, 756
CWE-120	Buffer copy without checking size of input	1493
CWE-119	Failure to constrain operations within the bounds of a buffer	7
CWE-118	Improper access of indexable resource	97

Figure 9: SAMATE test cases

RIPE [36]. This recently developed testsuite is designed to measure the coverage provided by buffer overflow defenses. It consists of 850 distinct exploits. Of these, 80 involve intra-object overflows. (Recall that such overflows are not detected by ours or other existing bounds checkers.) LBC detects a bounds violation with the remaining 770 exploits in this dataset.

Nullhttpd server. In this case, LBC detected two bound violations that were also detected by Baggy [5].

SAMATE Reference Dataset (SRD) [28]. The purpose of this reference dataset is "to provide users, researchers, and software security assurance tool developers with a set of known security flaws." It consists of test cases that illustrate various vulnerabilities catalogued in the Common Weakness Enumeration (CWE). Figure 9 lists all of test cases in the SRD on buffer-overflow-related CWEs. LBC was able to successfully handle all of these.

[9]In order for gcc to use an instrumented version of a function from `glibc`, one needs to prevent gcc from using its builtin version of the function by using `-fno-builtin-function` option.

5. Related Work

Comprehensive memory error detection approaches. Steffen's RTCC [34] was one of the early approaches to provide comprehensive detection of pointer errors. Its focus was on spatial errors, and the implementation approach was based on "fat" pointers that enlarge the size of a normal pointer to include additional metadata for memory error checking. Safe-C [6] is also based on fat pointers, but can detect temporal errors. The main difficulty with these approaches is that fat pointers change the representation of data (pointers as well as structures containing pointers) and are thus incompatible with uninstrumented libraries. Patil and Fischer [29] avoid the problems of fat pointers by storing metadata separately. Xu et al [38] improve on their results by providing support for unions and the most common form of type casts, and also providing a much faster implementation. SoftBounds [22] provides even better backward compatibility, supporting arbitrary type casts, and providing improved performance.

CCured [25] developed a novel static analysis to greatly reduce runtime overheads. However, it requires nontrivial porting effort to "cure" many large C-programs.

In summary, the main advantage of these techniques is that they can detect a larger class of memory errors than the bounds-checking techniques described below. Their drawback is that they experience significant compatibility problems, and much higher overheads.

Bounds checking techniques. This class of techniques do not maintain per-pointer metadata, and hence cannot answer the question of whether a given pointer is valid. Instead, they check each pointer arithmetic operation to ensure that it does not cause the pointer value to go out-of-bounds of its referent. Jones and Kelly [17] pioneered this technique. CRED [32] improved their work by providing much better compatibility with existing C-programs by providing a mechanism to represent out-of-bounds pointers.

Dhurjati et al [12] obtained significant speed improvements over CRED by leveraging the results of a whole-program analysis. They provide an improved representation for out-of-bounds pointers that avoids the need for checking them on each pointer dereference. However, the rest of the OOB object design remains the same as CRED, so they would share the problems described in the introduction related to OOB objects. Baggy [5] reports the fastest performance numbers among bounds checkers so far. Their performance comes from cleverly sizing objects in such a way that bounds checks can be performed very efficiently. Another source of performance improvement is their improved design of OOB pointers, but as discussed before, this design reduces compatibility with existing C-programs.

Paricheck [39] speeds up bounds-checking by attaching labels to objects, and verifying that pointer arithmetic does not lead to an object with a different label.

In summary, although bounds-checking provides improved performance and backward compatibility over techniques that maintain per-pointer metadata, their performance overheads are still significantly higher than LBC. Moreover, LBC does not break on programs that perform arbitrary pointer arithmetic and pointer-to-integer-to-pointer conversions. To achieve this level of compatibility, LBC trades off the ability to detect some bounds errors, specifically, those that "jump over" guard zones without accessing any data in between.

Security-targeted techniques. Rather than detecting memory errors like the above techniques, this class of techniques is aimed at detecting exploit attempts. The most popular among these is the ASLR technique that is widely deployed today. The technique provides excellent compatibility and performance, but is vulnerable to guessing attacks. This factor motivated LBC, which does not rely on protecting a secret. Moreover, LBC is fully compatible with ASLR and can be combined with it to provide even stronger protection. In particular, LBC, like previous bounds-checking techniques, does not provide any protection against intra-structure overflows. In contrast, ASLR, by virtue of randomizing pointer values, is resistant to the important subclass of intra-struct overflows that overwrite a pointer.

Similar to ASLR, DieHard [7] uses randomization to make it harder for attackers to exploit memory errors. Specifically, DieHard randomizes the location of objects in the heap, and the order in which freed objects are reused. The weakness of DieHard is that it offers probabilistic memory safety. LBC, in contrast, offers deterministic and prompt memory error detection.

Write-integrity testing (WIT) [4] uses a static analysis to identify all memory locations that can be written by an instruction, and assigns the same "color" to all these locations. Before a memory write, WIT ensures that the color associated with the write instruction matches the color of the location that is written. This approach enables the detection of out-of-bounds writes as well pointer-corruption due to intra-object overflows, subject to the limitations of a whole-program static (alias) analysis.

Since a static analysis may assign the same color to adjacent objects, WIT inserts a 8-byte region between objects that is given a color distinct from all valid objects. This idea is similar to our guard zone. However, LBC differs from and improves on WIT in several ways. First, LBC is optimized for checking both reads and writes, whereas WIT is focused on writes only. Although reads typically far outnumber writes, LBC's overhead is only about twice as much as WIT. Detecting out-of-bounds reads enables early detection of errors, and can support better error recovery. It can also prevent some information leakage attacks that rely on out-of-bounds reads.

Second, WIT always uses a fixed size region between objects, and hence can fail to detect overflows that involve arrays containing objects of size greater than 8 bytes. In contrast, LBC uses a guard zone size that is at least twice the size of array elements, and will detect overflows unless they are off by more than two elements.

Third, LBC is designed to maximize compatibility with arbitrary applications and third-party libraries. WIT, on the other hand, can run into compatibility problems with uninstrumented libraries. Moreover, it takes away a contiguous 12.5% of available address space (starting from address 0x40000000) for its metadata. In contrast, LBC's two-level structure for its metadata provides maximum compatibility with existing applications.

Data space randomization [9] associates a random mask with each data object, and uses the mask to randomize the representation of this object. Mask assignment uses an analysis similar to that used by WIT.

Debugging-oriented approaches. Tools such as Valgrind [26], Purify [15], mudflap [14] and mpatrol [31] are targeted at the software testing phase. Some of them, including Purify and mpatrol, use guard zones, but apply it only to heap objects. Moreover, they are typically slow, and are not intended for use on production software. Our contribution is the development of algorithms and techniques that enable guard zone operations to be efficient enough that they can be enabled on production code.

6. Conclusion

Out of bound arrays and pointers are the dominant source of memory errors in C programs. We presented a new lightweight backwards compatible approach for detecting these errors. While our design relies on the idea of guard zones that have been proposed before, our key contribution is the development of techniques to achieve excellent performance.

As opposed to some of the previous techniques, our approach achieves better backward compatibility and better runtime performance. On the CPU-intensive SPECINT benchmark, our overheads are less than half of the fastest times previously reported for bounds checking. Our technique achieves backward compatibility by avoiding pointer arithmetic checks, thereby coping with programs that perform arbitrary pointer manipulations and pointer-to-integer and integer-to-pointer conversions. Our evaluations demonstrated the ability of our technique to work seamlessly with uninstrumented libraries and modules, thus providing an easy migration path to using our approach. Furthermore, our emphasis on achieving good performance without breaking the separate compilation paradigm makes it easier to integrate the approach into the build processes of existing software. These factors make LBC a promising choice for applying to large collections of software, of the scale of operating system distributions, thus significantly decreasing the potential for exploitable memory corruption vulnerabilities in them.

7. References

[1] Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/.

[2] Internet Explorer IFRAME src & name parameter BoF remote compromise. http://www.kb.cert.org/vuls/id/842160, 2004.

[3] Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security*, 2010.

[4] Akritidis, Cadar, Raiciu, Costa, and Castro. Preventing memory error exploits with WIT. In *IEEE S&P*, 2008.

[5] Akritidis, Costa, Castro, and Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX security*, 2009.

[6] Austin, Breach, and Sohi. Efficient detection of all pointer and array access errors. *SIGPLAN*, 1994.

[7] Berger and Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *PLDI*, 2006.

[8] Bhatkar, DuVarney, and Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security*, 2003.

[9] Bhatkar and Sekar. Data space randomization. *DIMVA*, 2008.

[10] Bhatkar, Sekar, and DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security*, 2005.

[11] Checkoway, Davi, Dmitrienko, Sadeghi, Shacham, and Winandy. Return-oriented programming without returns. In *ACM CCS*, 2010.

[12] Dhurjati and Adve. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE*, 2006.

[13] Dhurjati and Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN*, 2006.

[14] Eigler. Mudflap: Pointer Use Checking for C/C++. In *GCC Developers Summit*, 2003.

[15] Hastings and Joyce. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. In *USENIX Winter Conference*, 1992.

[16] Jim, Morrisett, Grossman, Hicks, Cheney, and Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, 2002.

[17] Jones and Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Workshop on Automated Debugging*, 1997.

[18] Kendall. Bcc: run–time checking for C programs. In *USENIX Summer Conference*, 1983.

[19] Kil, Jun, Bookholt, Xu, and Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *ACSAC*, 2006.

[20] Lu, Li, Qin, Tan, Zhou, and Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[21] McPeak, Necula, Rahul, and Weimer. CIL: Intermediate language and tools for C program analysis and transformation. In *Compiler Construction*, 2002.

[22] Nagarakatte, Zhao, Martin, and Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *ACM PLDI*, 2009.

[23] Nagarakatte, Zhao, Martin, and Zdancewic. CETS: compiler enforced temporal safety for C. In *Symp. on Memory management*, 2010.

[24] Naraine. Adobe PDF exploits using signed certificates, bypasses ASLR/DEP. http://tinyurl.com/38kppsy, 2010.

[25] Necula, Condit, Harren, McPeak, and Weimer. CCured: type-safe retrofitting of legacy software. *ACM TOPLAS*, 2005.

[26] Nethercote and Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM PLDI*, 2007.

[27] NIST. NVD. http://nvd.nist.gov/.

[28] National Institute of Standards and Technology. SAMATE Reference Dataset Project. http://samate.nist.gov/SRD/.

[29] Patil and Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software — Practice & Experience*, 1997.

[30] PaX. Published on World-Wide Web at URL http://pax.grsecurity.net, 2001.

[31] Roy. Mpatrol. http://mpatrol.sourceforge.net/.

[32] Ruwase and Lam. A practical dynamic buffer overflow detector. In *NDSS*, 2004.

[33] Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM CCS*, 2007.

[34] Steffen. Adding run-time checking to the portable C compiler. *Software — Practice & Experience*, 1992.

[35] The MITRE Corporation. 2011 CWE/SANS Top 25 Most Dangerous Programming Errors. http://cwe.mitre.org/top25/.

[36] Wilander, Nikiforakis, Younan, Kamkar, and Joosen. RIPE: runtime intrusion prevention evaluator. In *ACSAC*, 2011.

[37] Xu, Bhatkar, and Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *USENIX Security*, 2006.

[38] Xu, DuVarney, and Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE*, 2004.

[39] Younan, Philippaerts, Cavallaro, Sekar, Piessens, and Joosen. PAriCheck: an efficient pointer arithmetic checker for C programs. In *ASIACCS*, 2010.