Checking Array Bound Violation Using Segmentation Hardware

Lap-chung Lam Tzi-cker Chiueh
Computer Science Department
Stony Brook University
{lclam, chiueh}@cs.sunysb.edu

April 15, 2005

Abstract

The ability to check memory references against their associated array/buffer bounds helps programmers to detect programming errors involving address overruns early on and thus avoid many difficult bugs down the line. This paper proposes a novel approach called *Cash* to the array bound checking problem that exploits the segmentation feature in the virtual memory hardware of the X86 architecture. The *Cash* approach allocates a separate segment to each static array or dynamically allocated buffer, and generates the instructions for array references in such a way that the segment limit check in X86's virtual memory protection mechanism performs the necessary array bound checking for free. In those cases that hardware bound checking is not possible, it falls back to software bound checking. As a result, *Cash* does not need to pay per-reference software checking overhead in most cases. However, the *Cash* approach incurs a fixed set-up overhead for each *use* of an array, which may involve multiple array references. The existence of this overhead requires compiler writers to judiciously apply the proposed technique to minimize the performance cost of array bound checking. This paper presents the detailed design and implementation of the *Cash* compiler, and a comprehensive evaluation of various performance tradeoffs associated with the proposed array bound checking technique. For the set of complicated network applications we tested, including Apache, Sendmail, Bind, etc., the latency penalty of *Cash*'s bound checking mechanism is between 2.5% to 9.8% when compared with the baseline case that does not perform any bound checking.

1 Introduction

Checking memory references against the bounds of the data structures they belong to at run time provides a valuable tool for early detection of programming errors that could have otherwise resulted in subtle bugs or total application failures. In some cases, these software errors might lead to security holes that attackers exploit to break into computer systems and cause substantial financial losses. For example, the buffer overflow attack, which accounts for more than 50% of the vulnerabilities reported in the CERT advisory over the last decade [3, 14, 18], exploits the lack of array bound checking in the compiler and in the applications themselves, and subverts the victim programs to transfer control to a dynamically injected code segment. Although various solutions have been proposed to subjugate the buffer overflow attack, inoculating application programs with strict array bound checking is considered the best defense against this attack. Despite these benefits, in practice most applications developers still choose to shy away from array bound checking

because its performance overhead is considered too high to be acceptable [13]. This paper describes a novel approach to the array bound checking problem that can reduce the array bound checking overhead to a fraction of the input program's original execution time, and thus make it practical to apply array bound checking to real-world programs.

The general problem of bound checking requires comparing the target address of each memory reference against the bound of its associated data structure, which could be a statically allocated array, or a dynamically allocated array or heap region. Accordingly, bound checking involves two subproblems: (1) identifying a given memory reference's associated data structure and thus its bound, and (2) comparing the reference's address with the bound and raising an exception if the bound is violated. The first subproblem is complicated by the existence of pointer variables. As pointers are used in generating target memory addresses, it is necessary to carry with pointers the ID of the objects they point to, so that the associated bounds could be used to perform bound checking. There are two general approaches to this subproblem. The first approach, used in BCC [4], tags each pointer with additional fields to store information about its associated object or data structure. These fields could be a physical extension of a pointer, or a shadow variable. The second approach [12] maintains an index structure that keeps track of the mapping between high-level objects and their address ranges, and dynamically searches this index structure with a memory reference's target address to identify the reference's associated object. The first approach performs much faster than the second, but at the expense of compatibility of legacy binary code that does not support bound checking. The second subproblem accounts for most of the bound checking overhead, and indeed most of the research efforts in the literature were focused on how to cut down the performance cost of address-bound comparison, through techniques such as redundancy elimination or parallel execution. At the highest compiler optimization level, the minimum number of instructions required in BCC [4], a GCC-derived array bound checking compiler, to check a reference in a C-like program against its lower and upper bounds is 6, two to load the bounds, two comparisons, and two conditional branches. In programs that involve many array references, this bound checking overhead could lead to serious performance penalty, even in the presence of various software optimizations. In this paper, we propose a new approach, called *Cash*[1], which exploits the segmentation support in the virtual memory hardware of Intel's X86 architecture [10] to perform array bound checking *without any per-reference overhead*. A segment in the X86 architecture can be of arbitrary size, ranging from a single byte to an entire address space. To provide inter-segment protection, X86's virtual memory hardware compares every memory reference with its associated segment's base address and limit, thus essentially checking it against the segment's lower and upper bounds. Recognizing the similarity between segment bound check and array bound check, *Cash* allocates a separate segment to each array, and generates the array reference instructions in such a way that the X86 architecture's segment bound checking hardware effectively performs the required array bound check for free. When array bound checking is done through segmentation hardware, there is no *per-reference overhead*. However, in some cases hardware bound checking is not possible, and *Cash* falls back to traditional software bound checking. Therefore, the overhead of *Cash* mainly comes from additional segments set-up required for hardware bound checking, and occasional software-based bound checking.

The general bound checking problem requires checking for each memory reference, including references to a field within a C-like structure. However, because the X86 architecture only supports a fixed number of segments (8192), allocating a segment to each object may quickly exhaust all the available segments. In addition, because *Cash* incurs a per-object set-up overhead, checking against bounds of non-array objects

[1] Checking Array bounds using Segmentation Hardware

may actually slow down the programs even more than the software approach. For these reasons, the current *Cash* prototype focuses only on bound checking for array-like references inside a loop, i.e., those of the form `A[i], A++, ++A, A--, or --A`, where `A` could be a pointer to a static array or a dynamically allocated buffer. For example, if a dynamic buffer is allocated through a `malloc()` call of the following form

```
X = (* TYPE) malloc(N * sizeof(TYPE))
```

where N is larger than 1, then *Cash* takes X as a pointer into an array of N elements, and *Cash* will check the references based on X if these references are used inside a loop. Because all known buffer overflow attacks take place in a loop context, focusing only on array-like references that are in a loop does not compromise *Cash*'s protection strength.

The rest of this paper is organized as follows. Section 2 reviews previous work on array bound checking and contrasts *Cash* with these efforts. Section 3 describes the detailed design decisions of the *Cash* compiler and their rationale. Section 4 presents a performance evaluation of the *Cash* compiler based on a set of array-intensive programs, and a discussion of various performance overheads associated with the *Cash* approach. Section 5 concludes this paper with a summary of the main research ideas and a brief outline of the on-going improvements to the *Cash* prototype.

2 Related Work

Most previous array bound checking research focused on the minimization of run-time performance overhead. One notable exception is the work from the Imperial College group [12], which chose to attack the reference/objection association problem in the presence of legacy library routines. The general approach towards optimizing array bound checking overhead is to eliminate unnecessary checks, so that the number of checks is reduced. Gupta [15, 16] proposed a flow analysis technique that avoids redundant bound checks in such a way that it still guaranteed to identify any array bound violation in the input programs, although it does not necessarily detect these violations immediately after they occur at run time. By trading detection immediacy for reduced overhead, this approach is able to hoist some of the bound checking code outside the loop and thus reduce the performance cost of array bound checking significantly. Asuru [11] and Kolte and Wolfe [13] extended this work with more detailed analysis to further reduce the range check overhead.

Concurrent array bound checking [6] first derives from a given program a reduce version that contains all the array references and their associated bound checking code, and then runs the derived version and the original version on separate processors in parallel. With the aid of a separate processor, this approach is able to achieve the lowest array bound checking overhead reported until the arrival of *Cash*.

Unlike most other array bound checking compiler projects, the Bounds Checking GCC compiler (BCC) checks the bounds for both array references and general pointers. Among the systems that perform both types of bound checks, BCC shows the best software-only bound checking performance. However, BCC only checks the upper bound of an array when the array is accessed directly through the array variable (not pointer variable) while *Cash* automatically checks both the upper and lower bounds. Since the *Cash* compiler is based on BCC, it can also check the bounds for general pointers.

The array bound checking problem for Java presents new design constraints. Because bound checking code cannot be directly expressed at bytecode level, elimination of bound checks can only be performed at run time, after the bytecode program is loaded. Directly applying existing array bound checking optimizers

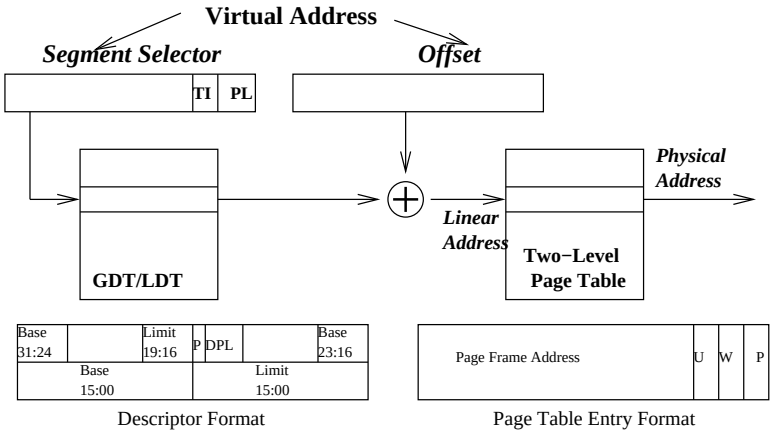

Figure 1: *Memory translation process in the X86 architecture's virtual memory hardware*

at run time is not feasible, however, because they are too expensive for dynamic compilation. ABCD [17] is a light-weight algorithm for elimination of Array Bounds Checks on Demand, which adds a few edges to the SSA data flow graph and performs a simple traversal of the resulting graph. Despite its simplicity, ABCD has been proven quite effective. Xi and Pfenning [7,8] developed a type-based approach to eliminating array bound checking and list tag checking by conservatively extending Standard ML with a restricted form of dependent types. This enables the programmer to capture more invariants through types while type-checking remains decidable in theory and can still be performed efficiently in practice.

Electric Fence [1] is a malloc debugger for Unix like system, which can only bound check the dynamic buffers. It places an inaccessible memory page immediately after or before each memory region allocated. When program reads or writes these inaccessible pages, virtual memory hardware issues a segmentation fault, stopping the program at the offending instruction. Although this method has zero overhead to perform array bound checking, it consumes too much virtual memory space, which could lead to excessive page faults and cache misses.

Intel X86 architecture includes a bound instruction [9] for array bound checking. However, the bound instruction is not widely used because on 80486 and Pentium processors, the bound instruction is slower than the six normal equivalent instructions. The bound instruction requires 7 cycles on a 1.1 GHz P3 machine while the 6 equivalent instructions require 6 cycles. Although the segmentation hardware feature has existed since the 386 days, this architectural feature was never exploited by the operating systems or the compiler writers. A conspicuous exception is the Palladium system [19], which exploits segmentation hardware for intra-address space protection, which achieves the lowest inter-protection domain control transfer overhead that has ever been reported in the literature. Glen Pearson described a programming technique for programmers to exploit segmentation hardware in a way similar to *Cash* to add array bound checking to DOS applications [5]. Glen Pearson replaced the Turbo C malloc library with his own library, which allocates a new segment for each dynamically allocated buffer. However, this method cannot deal with the arrays not allocated through malloc since it requires the modification of the compiler. According to our best knowledge, *Cash* is the only C compiler that utilizes the segmentation hardware to optimize the bound checking of both static and dynamic allocated arrays.

3 The Cash Approach

3.1 Segmentation-Based Virtual Memory Support in the X86 Architecture

Before describing the detailed design of the *Cash* compiler, let's briefly review the X86 architecture's virtual memory hardware. Intel X86 architecture's virtual memory hardware supports both variable-length segments and fixed-sized pages, as shown in Figure 1. A virtual address consists of a 16-bit *segment selector*, which resides in one of the six on-chip segment registers, and a 32-bit *offset*, which is given by `EIP` register for instruction references, `ESP` register for stack operations, or other registers/operands in the case of data references. The segment selector contains a 13-bit index into the *Global Descriptor Table* (GDT) or the current process's *Local Descriptor Table* (LDT). The choice between GDT and LDT is determined by a TI bit in the segment selector. Each process has its own LDT whereas the GDT is shared among processes. The number of entries in each LDT and the GDT is 8192. The GDT or LDT entry indexed by the segment selector contains a *segment descriptor*, which, among other things, includes the base and limit addresses of the segment, the segment's descriptor privilege level (DPL), and read/write protection bits. The 32-bit offset is added to the associated segment's start address to form a 32-bit *linear address*. The most significant 20 bits of a linear address are a virtual memory page number and are used to index into a two-level page table to identify the corresponding physical page's base address, to which the remaining 12 bits are added to form the final physical address. The page size is 4 Kbytes.

The first entry of the GDT is not used by the X86 architecture. A segment selector that points to this entry of the GDT (that is, a segment selector with an index of 0 and the TI flag set to 0) is used as a null segment selector. The processor does not generate an exception when a segment register (other than the code and stack segment registers) is loaded with a null selector. It does, however, generate an exception when a segment register holding a null selector is used to access memory. A null selector can be used to initialize unused segment registers, so that use of segment registers that have not been properly initialized could be caught at run time. Loading the code or stack segment register with a null segment selector causes a general-protection exception to be generated. A LDT is itself a segment whose segment descriptor is in the GDT. If there are multiple LDTs, each must have a separate segment selector and segment descriptor in the GDT. To eliminate address translation overhead when accessing an LDT, the segment selector, base address, limit, and access rights of the LDT are stored in the LDTR register.

Intel X86 architecture provides protection checks at both segment and page levels. When a linear address is formed, the hardware checks whether it is within the corresponding segment's range as specified in the segment descriptor's base and limit. That is, the segment limit check checks both the upper and lower bounds of a segment. In addition to checking segment limits, the processor also checks descriptor table limits. The GDTR and LDTR registers contain 16-bit limit values that the processor uses to prevent programs from selecting a segment descriptor outside the GDT and LDT, respectively. In addition, these two registers also contain the base addresses of the GDT and the LDT, which are used to compute the addresses of target GDT or LDT entries. In addition to limit check, there are standard protection mechanisms based on segment or page privilege levels: program execution based on code residing at a less privileged level cannot access data segments or jump to code segments that are at a more privileged level.

Every segment register has a visible part and a hidden part. The hidden part is sometimes referred to as a descriptor cache or a shadow register. When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. The

information cached in the segment register (visible and hidden) allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor. In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. If this is not done, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified. The `MOV` instruction can be used to load a segment register as well as store the visible part of a segment register into a general-purpose register.

3.2 Mapping from References to Objects

To check whether a memory reference exceeds its bound, one needs to determine the high-level data structure or object with which the reference is associated. *Cash* solves this reference-object association problem by using a shadow pointer approach. Each pointer variable P is augmented with another pointer P_A to an information data structure about the object to which P points. P and P_A thus form a new structure on its own. which is still pointed by P. Both P and its P_A are copied in all pointer assignment/arithmetic operations, including binding of formal and actual pointer arguments in function calls. Because P and P_A are guaranteed to be adjacent to each other, the *Cash* compiler can easily identify the high-level object with which a memory reference is associated by following the pointer variable used to generate the reference's effective address. Here we are assuming that each array reference is of the form of a base address plus an offset, and the location following the variable holding the base address contains the array's information structure.

The information structure to which P_A points contains three words, the lower and upper bounds of the object's address range, and the LDT index associated with the segment allocated to the object. Although it is possible to retrieve the base and limit of an object using the associated segment's LDT index, *Cash* maintains the object's base and limit explicitly in the per-object information structure because it reduces the memory access overhead of software-based bound checks. The three-word per-object information structure of an array is allocated when the array is created. For example, when a 100-byte array is statically allocated, *Cash* allocates 112 bytes, with the first three words dedicated to this array's information structure. The same thing happens when an array is allocated through `malloc()`.

3.3 Array Access Code Generation

To exploit X86's segment limit check hardware, one needs to generate instructions for array references so that the implicit segment limit check that occurs in each memory access does exactly what array bound checking requires. This involves two operations. First, every time an array is used, *Cash* needs to allocate a segment register for the array, and initializes the segment register with the array's associated segment selector. Because both operations are loop independent, no additional instructions are required inside a loop containing the array references. Second, the offset of each array reference needs to be recomputed using the start of the array as the basis. The following shows the assembly code for an example C statement involving array references, `A[i] =10`, where A is a pointer to an array.

```
    Without Array Bound Check

movl    -60(%ebp), %eax    ; load i
```

```
leal    0(, %eax, 4), %edx ; i * 4
movl    -56(%ebp), %eax    ; load a
movl    $10, (%edx, %eax)  ; mem[a+4*i] = 10
```

Checking Array Bound using Cash

```
movl    -60(%ebp), %eax     ; # load i
leal    0(, %eax, 4), %edx  ; i * 4
movl    -56(%ebp), %eax     ; # load a
movw    -52(%ebp), %ecx     ; # load a's shadow
                            ; structure ptr
movw    0(%ecx), %gs        ; load GS
subl    4(%ecx), %eax       ; compute offset
movl    $10, %gs:(%edx,%eax); check bounds and mem[a+4*i]=10
```

If the `A[i] = 10` statement is inside a loop, then a standard optimization compiler can move the three instructions in the *Cash* version that are marked with # outside the loop. Because a single segment register loading instruction takes 4 cycles, it is essential that segment register loading is done outside the outermost loop. Consequently, the bound checking in *Cash* really does not incur any extra software overhead per array reference as compared with the version without array bound check.

3.4 Segment Allocation and Deallocation

In *Cash*, when an array is created, the associated segment is also allocated. Similarly, when an array is freed, the associated segment is deallocated. If an array is allocated statically (global array), the *Cash* compiler inserts code into the beginning of the program to perform segment initialization, which includes allocating a segment, setting up the segment's LDT entry, and filling in the segment's three-word information structure accordingly. Each local array in a function requires a separate segment initialization step inserted into the function prologue, and a segment clean-up step into the function epilogue. For dynamically allocated arrays, the `malloc()` and `free()` routines are modified to include the segment initialization and clean-up steps, respectively. *Cash* does not change the way that GCC allocates memory for global or local variables. A segment is created on the top of the original memory region of an array variable. The segment base is the beginning address of the array, and the limit is the size of the array.

In a 16-bit segment selector, only 13 bits of them are used as an index into the LDT. Therefore, at most 8192 segments can exist in a program. Excluding the first entry, which is used to store a call gate, there are only 8191 segments left for array bound checking. In case there are more than 8191 objects that need to co-exist simultaneously in a program, the *Cash* compiler assigns a global segment to those objects for which no free segments are available, essentially disabling the array bound checking for these objects. The global segment is the original application data segment created by the Linux kernel. An alternative approach to this problem is to allocate multiple LDTs per process, and dynamically change the LDTR to point to a particular LDT to at run time. However, modifying the LDTR requires a system call, and may lead to thrashing in the form of LDT switching.

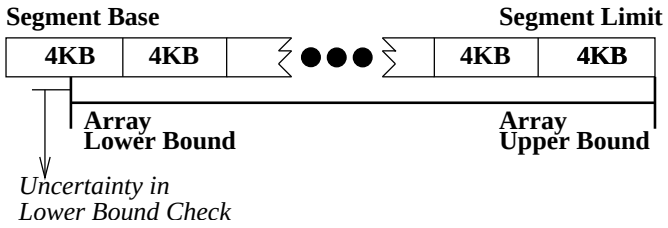

Figure 2: *The lower bound check of* Cash *is not 100% strict when the segment allocated for an array is larger than 1 Mbytes. The uncertainty is as large as 1 page or 4Kbytes.*

3.5 Segment Size Consideration

The *limit* field of a segment descriptor is 20 bits wide. To support large segments, the X86 architecture includes a *granularity* bit in the segment descriptor, which, when turned on, scales the unit of the segment size by a factor of 2^{12}. That is, when the *granularity* bit is off, the segment size ranges from 1 to 2^{20} bytes or 1 Mbytes. However, when the *granularity* bit is set, the segment size ranges from 2^{12} or 4Kbytes to 2^{32} or 4Gbytes. Therefore, to support large arrays, the *granularity* bit needs to be turned on.

However, when the *granularity* bit is turned on, the X86 architecture's segment limit check will ignore the least significant 12 bits of the offset part of a virtual address. Consequently the bound checking may no longer be 100% strict. To address this problem, for an array whose size is larger than 1 Mbytes, the *Cash* compiler always allocates a segment whose size is the minimum multiple of 4Kbytes that is larger than or equal to the target array's size. In addition, *Cash* aligns the end of the target array with the end of the allocated segment. With this set up, the segment limit check hardware performs the upper array bound check correctly, i.e., down to individual bytes, but the lower array bound check may be off by at most one page. Figure 2 illustrates how this technique provides an upper bound check for large arrays whose size is larger than 1 Mbytes. In the ideal case, if there are no other data structures that are allocated from the portion of the first page of such a segment that does not belong to the associated array, then the deficiency in lower bound check practically does no harm. However, the current *Cash* compiler is built in such a way that it layers segments on top of arrays that are already allocated by the underlying compiler (in this case GCC), and therefore does not have any control over data structure placement in the address space.

In practice, this lack of strict lower bound check does not impose any security risk, because (1) all known buffer overflow attacks overflow an array/buffer through the upper bound, and (2) with additional compiler support the area between the array's lower bound and the lowest page's boundary could be forced to be unused, and therefore cannot contain anything useful to be corrupted. Moreover, applications rarely use arrays larger than 1 Mbytes in practice. None of the 18 applications we tested use arrays whose size is larger than 1 Mbytes.

3.6 OS Support

To allocate a segment for an array, a segment descriptor must be inserted into the LDT table. However, because the per-process LDT is in the kernel space, modifying the LDT requires a system call. If a function contains local arrays, this means that calling such a function requires a system call for each of its local arrays to set up the array's associated segment. The system call `modify_ldt()` of the Linux OS is used to add or remove an entry from the LDT. This system call on a 1.1 GHz PIII machine requires 781 clock cycles.

This is obviously an unacceptable overhead. Our original solution was to move the LDT to the user address space. However, moving LDT to the user space creates a security problem, because the LDT is also used to store call gates that attackers can exploit to access the kernel space. Therefore, we eventually still keep the LDT inside the kernel, and develop several optimizations to reduce the overhead of LDT modification to the minimum.

The first optimization is a low-overhead system call mechanism to modify the LDT inside the kernel. On the X86 architecture, there are at least two ways for user applications to access the kernel code, through the `int` instruction or a call gate using the `lcall` instruction. Linux 2.4 kernel uses the `int 0x80` instruction to implement system calls. *Cash* introduces a new system call `set_ldt_callgate(void)` to set up a call gate that points to a new kernel function called `cash_modify_ldt()`, which modifies the LDT. When an application compiled by *Cash* is executed, it first calls this system call to install the call gate in the first entry of the LDT, and subsequently uses the `lcall $0x7, $0x0` instruction to call `cash_modify_ldt()` for LDT modification. Unlike a normal Linux system call, which saves all registers at the beginning and restores them before the system call returns, `cash_modify_ldt` only saves the EDX and the DS register. All parameters are passed to this function through registers to eliminate the overhead of copying the parameters from the user stack to the kernel stack. As a result, `cash_modify_ldt` only takes 253 clock cycles while `modify_ldt` system call takes 781 clock cycles.

The second optimization is to perform LDT entry allocation and de-allocation in user space. Each application keeps a `free_ldt_entry` list in user space to maintain all free LDT entries. When a segment is freed, a user application does not need to go into the kernel to modify the LDT, it only puts the segment's LDT entry number back to this `free_ldt_entry` list, and the LDT entry can be reused next time for a new segment.

In many cases, a function containing local array references is used inside a loop. Each time the function is called, it needs to allocate one LDT entry for each local array. To avoid redundant modification to the LDT in these cases, *Cash* keeps a 3-entry cache to store the three most recently freed segments. When a new segment is needed and its base and limit match one of three segments in the cache, *Cash* simply reuses the matched free segment and avoids the overhead of going into the kernel to modify the LDT. Since freeing a segment never modifies the LDT, *Cash* can safely reuse the matched entry. This optimization dramatically reduces the frequency of LDT modification for functions that contain local arrays and are called many times inside loops.

3.7 Segment Register Allocation

To reduce address translation time and coding complexity, the X86 architecture provides six segment registers to hold segment selectors. To access a segment requires the corresponding segment selector to be loaded into one of the segment registers. That is, although a program can define up to 8192 segments, only six of them are available for immediate use. Three of these six segment registers, CS, SS, and DS, are are reserved for code, stack, or data references, respectively. The three other segment registers, ES, FS, and GS, are available for loading additional segments. The current *Cash* prototype only uses ES, FS, and GS segment registers for array bound checking. Our experiments indicate that three segment registers are sufficient for most network applications. *Cash* allocates segment registers on a first-come-first-serve basis. The first three arrays the *Cash* compiler encounters during the parsing phase inside a (possibly nested) loop are assigned one of the three segment registers. If more than three arrays are involved within a loop, *Cash* falls back to software array bound checking for references associated with those arrays beyond the first three.

If a segment register is used for array bound checking in a function, the current value of that segment register must be saved at the function entry and then restored before the function returns. For example, ES is used by string instructions, which currently only hand-crafted assembly code use. Therefore, to free ES, all GLIBC string functions, which are written in assembly code, are manually modified so that the current ES value is saved at the function entry, ES is loaded with the same value as DS, and the saved value is restored at the function exit. Similarly if a function uses a segment register, *Cash* inserts code in the function prologue to save the current value, and restore it back at the function epilogue.

With more segment registers, it is less likely that *Cash* invokes software bound checking, and the performance overhead can be reduced further. We have successfully used 4 segment registers (ES, FS, GS, and SS) on a set of simple numerical kernel programs, and the resulting performance, which is reported in Section 4.2, is indeed better. Linux initializes the SS segment register with the same value as the DS register. It is therefore tempting to consolidate them into one segment register. However, the DS register is used in data reference and therefore cannot be used for other purposes. The SS register, on the other hand, is used only in PUSH/POP instructions, and it is the default segment register for EBP and ESP registers. To use SS in array bound checking, *Cash* replaces PUSH/POP instructions with normal MOVE and SUB/ADD instructions, and substitutes the DS segment register for the SS register in each instruction involving the EBP/ESP register. From our experiments, replacing PUSH/POP with MOVE and SUB/ADD does not seem to incur any performance penalty. As an example, for the following C function,

```
void foo(int a, int b)
{
  int c = a+b;
  printf("%d\n",c);
}
```

we can generate a code sequence that does away with PUSH/POP instructions, and forces all instructions involving EBP/ESP to use DS rather than SS, as follows:

```
      ORIGINAL                    MODIFIED
foo:
1 pushl   %ebp              1a subl    $4, %esp
                            1b movl    %ebp, %ds:(%esp)
2 movl    %esp, %ebp        2a movl    %esp, %ebp
3 subl    $8, %esp          3a subl    $8, %esp
4 movl    12(%ebp), %eax    4a movl    %ds:12(%ebp), %eax
5 addl    8(%ebp), %eax     5a addl    %ds:8(%ebp), %eax
6 movl    %eax, -4(%ebp)    6a movl    %eax, %ds:-4(%ebp)
7 subl    $8, %esp          7a subl    $8, %esp
8 pushl   -4(%ebp)          8a subl    $4, %esp
                            8b movl    %ds:-4(%ebp), %ecx
                            8c movl    %ecx,%ds:(%esp)
9 pushl   $.LC0             9a subl    $4, %esp
                            9b movl    $.LC0,%ds:(%esp)
10 call    printf           10a call    printf
```

```
11 addl    $16, %esp       11a addl    $16, %esp
12 leave                   12a leave
13 ret                     13a ret
```

While it is possible to free SS as described above, it involves significant change to the compiler's code generator. Therefore the current *Cash* prototype still uses three segment registers by default.

Since all existing operating systems assume a flat address space model, CS, DS, and SS registers are initialized with the same segment selector, and therefore contain redundant information. It is therefore tempting to consolidate them into one segment register. However, the CS register cannot be eliminated because the instruction fetch process uses the contents of the CS register; the DS register is used in every data references and therefore cannot be used for other purposes, either. The SS register, on the other hand, is used only in `PUSH` and `POP` instructions, and can be made available for array bound checking if the *Cash* compiler replaces `PUSH` and `POP` with the normal `MOVE` instruction. In summary, the *Cash* compiler has up to four segment registers at its disposal for simultaneous use. If more than four arrays are involved within a loop, the *Cash* compiler resorts to software array bound checking for all arrays beyond the fourth array.

3.8 Security Consideration

Applications compiled by BCC are more secure than *Cash* applications since BCC bound-checks all pointers. However, the high performance overhead associated with BCC applications prevents them being used in production mode. The weakness of *Cash* is that *Cash* does not check pointers and array references outside loops. However, almost all known attacks that exploit bound violation vulnerability involve array references inside loops. Further, the low overhead of *Cash* makes it more likely to be used in real world. Currently *Cash* bound-checks both read and write operations since it is designed for the purposes of both security and debugging. If *Cash* is used for security only, *Cash* does not need to bound-check read operations and thus can further decrease the performance overhead by reducing the number of segment register required and and the number of software bound checks.

In *Cash*, the `free_ldt_entry` list could be corrupted due to a program bug because it is in user space. However, the worst damage as a result of this corruption is to crash the application itself, but will not affect other processes or the OS because the LDT table is not shared among processes. The descriptors of the per-process code, data, stack segments are stored in the GDT, and therefore will not be affected by LDT modification. Our own experiences confirmed that the LDT entries never get modified under Linux by any code other than *Cash*. Finally, the kernel function `cash_modify_ldt` guarantees that no call gate and privilege segment can be created in the LDT.

3.9 Miscellaneous Issues

One of the major limitations of the *Cash* compiler is that it does not work with the binary code of legacy libraries, because of the use of multi-word pointer representation. However, this limitation is shared by all known array bound checking compilers except the work from Kelly and Jones [12]. The only solution to this problem is to re-compile the commonly used library functions. However, many functions in GLIBC are implemented in assembly code. To correctly pass parameters into and get the results back from these functions, we manually insert assembly macros into these functions to calculate parameter address and

to do bound checking. All system call stubs in GLIBC are modified since system calls only take one-word pointers. With these modifications, the *Cash* prototype is able to compile the modified GLIBC2.2.2 successfully

Cash chooses not to allocate a separate segment for each scalar variable to conserve segments. For the statement like `p = &a` where `p` is an integer pointer, and `a` is an integer scalar variable, *Cash* associates `p` to the global segment, essentially turning off bound checking for `p`. As a result, *Cash* does not bound-check array pointers resulting from type casting of another pointer. For example, after the statement `q = (char *) p`, where `p` is an integer pointer and `q` is a character pointer, *Cash* could have treated `q` as an array pointer. However, because *Cash* does not bound-check `p`, copying `p`'s shadow information structure to q's shadow information structure also disables bound checking for `q`. While this is a limitation, we believe this should not pose real problems in practice because such type casting is rarely used.

A common misconception about *Cash* is that it requires alias analysis. *Cash* does not need to know which pointers point to which objects statically, because it simply generates code to pass the object bound and reference information around as part of the pointer arithmetic and assignment operations, without concerning the pointer values.

Because *Cash* only adds a segment layer on top of an otherwise flat address space, and does not modify the internals of malloc, *Cash* does not introduce any additional fragmentation beyond what exists in the original malloc. In *Cash*, there will not be any unused space within a segment because a segment can start and end at arbitrary byte boundaries.

Program Name	**HW/SW Checks**	**GCC**	***Cash***	**BCC**
SVDPACKC	403/0	5291993K	1.8%	120.0%
Vol. Render.	45/0	425029K	3.3%	126.4%
2D FFT	13/0	25870K	3.9%	72.2%
Gaus. Elim.	45/0	46961K	1.6%	92.4%
Matrix Multi.	14/0	62861K	1.5%	143.8%
Edge Detect	137/0	806514K	2.2%	83.8%

Table 1: *The performance comparison among GCC, BCC, and* Cash *based on a set of kernels that use array references extensively. The matrix used in SVDPACKC (singular value decomposition) is 374x82, the data set used in the Vol. Render. (Volume Renderer) is 128x128x128 and the image plane's resolution is 256x256, the resolution of the 2D images used in the 2D FFT run is 64x64, the matrix used in Gaus. Elim. (Gaussian Elimination) and Matrix Multi. (Matrix Multiplication) is 128x128, the resolution of the 2D image used in Edge Detect (Image Edge Detection) is 1024x768. All performance measurements are in terms of thousands of CPU cycles. The numbers inside the parentheses represent additional execution time due to array bound checking in percentage with respect to GCC.*

4 Performance Evaluation

4.1 Prototype Implementation

The current *Cash* compiler prototype is derived from the Bounds Checking GCC [4], which is derived from GCC 2.96 version, and runs on Red Hat Linux 7.2. We chose BCC as the base case for the two reasons. BCC is one of the most advanced array bound checking compilers available to us, boasting a consistent performance penalty of around 100%. It has been heavily optimized. The more recent bound checking performance study from University of Georgia [2] also reports that the average performance overhead of BCC for a set of numerical kernels is around 117% on Pentium III. In contrast, all the research results published in the literature on C-based array bound checking *always* did far worse than BCC. Moreover, the fact that BCC and Cash are based on the same GCC code basis makes the comparison more meaningful. Existing commercial products such as Purify are not very competitive. Purify is a factor of 5-7 slower than the unchecked version because it needs to perform check on every read and write. The VMS compiler and Alpha compiler also supported array bound checking, but both are at least twice as slow compared with the unchecked case on the average. In all the following measurements, the compiler optimization level of both BCC and *Cash* is set to the highest level. All test programs are statically linked with all required libraries, which are also recompiled with *Cash*.

Instead of a 3-word pointer representation in BCC, *Cash* uses a 2-word pointer representation, with an additional 3-word shadow information structure. This change is motivated by the observation that pointer variable manipulation may result in substantial memory copying overhead when pointer size is increased by a factor of three. Moreover, rather than checking bound violation for every array reference via software, *Cash* performs these checks mostly through the segment limit checking hardware supported on the X86 architecture. To move the per-process LDT from the kernel address space to the user address space, *Cash* introduces a new system call to the Linux operating system, which is called in the beginning of a bound-checked program to modify the LDTR to point to a table allocated in the user address space.

To understand the quantitatively results of the experiments run on the *Cash* prototype presented in the next subsection, let's first analyze qualitatively the performance savings and overheads associated with the *Cash* approach.

Compared with BCC, *Cash*'s bound checking mechanism does not incur any software overhead, because it exploits segment limit check hardware to perform array bound checks for free. However, there are other overheads that exist only in *Cash* but not in BCC. First, there is a *per-program overhead*, which results from the set-up of the call gate and the segment free list. Then there is a *per-array overhead*, which is related to segment allocation and deallocation. Finally there is a *per-array-use overhead*, which is due to segment register loading whenever an array is to be used. Because an array may be used at different points of program execution, each use may incur a segment register loading overhead if its corresponding segment selector is not loaded into any of the available segment registers. On a Pentium-III 1.1-GHz machine running Red Hat Linux 7.2, the measured *per-program overhead* is 543 cycles, the *per-array overhead* is 263 cycles, and the *per-array-use overhead* is 4 cycles.

4.2 Micro-Benchmark Results

To study the array bound checking overhead associated with *Cash*, we first choose a set of six numerical kernels that use array references extensively, and the performance results in thousands of CPU cycles are

shown in Table 1. We compare *Cash* against the vanilla GCC without bound checking and the bound-checking version of GCC (BCC). The former sets the baseline performance, whereas the latter represents a state-of-the-art fully operational software-only bound checking compiler. SVDPACKC is a C-based singular value decomposition package that implements Lanczos and subspace iteration-based methods for determining several of the largest singular triplets (singular values and corresponding left- and right-singular vectors) for large sparse matrices. The Volume Rendering program implements a ray casting algorithm for 3D volumetric data sets. The 2D FFT, Gaussian Elimination, Matrix Multiplication, and Image Edge Detection programs are based on standard algorithms. These programs represent the best case for BCC because they use pointers rarely, and they are more amenable to advanced compiler optimizations.

We instrumented the *Cash* compiler to measure the number of bound checks statically inserted into the applications when BCC is in use, and how may of them are eliminated by the *Cash* compiler and replaced with hardware checks. The second column of Table 1 shows the number of hardware bound checks and software bound checks of each program. In this experiment, *Cash* is able to use four segment registers. As a result, *all* software bound checks are eliminated in each of the six test programs. In terms of execution time, the performance overhead of the *Cash* compiler is always within 4% of that of GCC, for all six test programs. In contrast, BCC is between 1.7 to 2.4 times slower than GCC. Although BCC's performance is already quite good compared with results reported in the literature, it is still significantly slower than *Cash*.

However, if there are only 2 segment registers available, only in Volume Rendering, 2D FFT, and Gaussian Elimination can all software bound checks be eliminated. For SVDPACKC, Matrix Multiplication, and Image Edge Detection, the percentages of software bound checks that can be eliminated are 50.1% (202/201), 85.7% (12/2), and 19.7% (27/110), respectively. Accordingly, their overall performance overheads become 35.7%, 1.5%, and 44.2%, respectively. The reason why software bound checks are needed in these programs is that the programs access more than two arrays at a time and references to those arrays for which no segment register can be assigned have to be bound-checked through software. As expected, the higher the percentage of array references that are bound-checked through software, the larger *Cash*'s performance overhead.

Program Name	**GCC**	***Cash***	**BCC**
SVDPACKC	421,076	29.9%	127.1%
Volume Rendering	379,316	30.1%	124.2%
2D FFT	393,332	28.6%	135.9%
Gaussian	364,788	29.8%	125.6%
Matrix	363,540	29.9%	145.2%
Image Edge	375,348	30.4%	146.5%

Table 2: *The binary code size comparison among GCC, BCC, and* Cash *for the test suite. GCC numbers are in bytes and numbers for* Cash *and BCC are in terms of percentage increases with respect to GCC. Programs are compiled with static linking.*

There are two reasons why the binary size of a program with bound checking is larger than one without bound checking. First, the bound checking instructions take additional space. Second, pointer representation require multiple words. BCC incurs both overheads, whereas *Cash* only needs to bear the second cost. Table 2 shows that the code size overhead of the applications compiled by the *Cash* compiler is within 31% of that of GCC, whereas BCC's code is more than 120%.

Program Name	64	128	256	512
2D FFT	3.9%	1.5%	0.1%	0.001%
Gaussian	5.7%	1.6%	1.7%	0.3%
Matrix	2.2%	1.5%	1.4%	0.1%

Table 3: *The relative performance cost of* Cash *with respect to GCC for 2D FFT, Gaussian Elimination, and Matrix Multiplication decreases as the input matrix size increases. 64 stands for 64x64 matrix, 128 stands for 128x128 matrix, etc.*

An important property of the *Cash* approach is that its absolute (not relative) overhead is independent of the size of the data set that the application programs are manipulating, if all software array bound checks are replaced with hardware checks. Therefore, the relative overhead of *Cash* compared to GCC should decrease as the data set size increases. Table 3 shows that the relatively performance cost of *Cash* compared with the vanilla GCC in general decreases as the matrix size increases. However, the trend is not absolutely monotonic because large inputs cause higher cache miss ratio and thus complicate the relative performance cost calculation. However, this general trend reflects that the performance overhead of *Cash* is indeed independent of the data set size and thus is more scalable than software bound checkers.

Program Name	Lines of Code	Brief Description	Array-Using Loops	$>$ 3 Arrays
Toast	7372	GSM audio compression utility	51	6 (0.6%)
Cjpeg	33717	JPEG compression utility	236	38 (1.5%)
Quat	15093	3D fractal generator	117	19 (3.4%)
RayLab	9275	Raytracer-based 3D renderer	69	4 (0.2%)
Speex	16267	Voice coder/decoder	220	23 (2.8%)
Gif2png	47057	Gif to PNG converter	277	9 (1.3%)

Table 4: *Characteristics of a set of large applications used for the macro-benchmarking study. The source code line count includes all the libraries used in the programs, excluding* `libc`.

Program Name	GCC	*Cash*	BCC
Toast	4,727,612K	4.6%	47.1%
Cjpeg	229,186K	8.5%	84.5%
Quat	9,990,571K	15.8%	238.3%
RayLab	3,304,059K	4.5%	40.6%
Speex	35,885,117K	13.3%	156.4%
Gif2png	706,949K	7.7%	130.4%

Table 5: *The performance comparison among GCC, BCC, and* Cash *based on a set of macro-benchmark programs. GCC numbers are in thousands of CPU cycles, whereas performance penalty of* Cash *and BCC are in terms of execution time percentage increases with respect to GCC.*

4.3 Macro-Benchmark Results

We also compare the performance of GCC, BCC, and *Cash* using a set of large applications, whose characteristics are listed in Table 4, and the results are shown in Table 5. In general, the performance difference between *Cash* and BCC is smaller compared with the results in Table 1 because the total cost of array bound checking is relatively less significant when the original program size is large. However, even for these applications, the performance overhead difference between BCC and *Cash* is still quite substantial. As for binary code size, the results for the macro-benchmark suite, shown in Table 6, are similar to that for the micro-benchmark suite.

Program Name	GCC	*Cash*	BCC
Toast	476,600	61.8%	123.5%
Cjpeg	476,376	52.5%	130.9%
Quat	523,096	58.9%	151.2%
RayLab	501,048	35.8%	130.8%
Speex	530,584	30.6%	136.9%
Gif2png	516,664	35.8%	136.6%

Table 6: *The binary code size comparison among GCC, BCC, and* Cash *for the macro-benchmark suite. GCC numbers are in bytes and numbers for* Cash *and BCC are in terms of percentage increases with respect to GCC. Programs are compiled with static linking.*

A major concern early in the *Cash* project is that the number of segment registers (currently 3) is so small as to cause frequent fall-back to software bound check. Because *Cash* only checks array references within loops, a small number of segment registers is a problem only when the body of a loop uses more than 3 arrays/buffers. That is, the limit on the number of simultaneous array uses is per loop, not per function, or even per program. To isolate the performance cost associated with this problem, we measure the number of loops that involve array references, and the number of loops that involve more than 3 distinct arrays (called *spilled loops*) during the execution of the micro-benchmark programs. The results are presented in Table 4, which show that the majority of array-referencing loops in these programs use fewer than 5 arrays. The percentage numbers within the parenthesis provide the percentage of loop iterations that are executed in the experiments and that belong to spilled loops. However, the fact that some loops do use more than 3 distinct arrays simultaneously suggests that these programs need to incur software bound checking overhead, and is another reason why the performance overhead of *Cash* is higher for the macro-benchmark suite than the micro-benchmark suite.

4.4 Network Applications

Because one of the applications of bound checking is to stop remote attacks that that exploit buffer overflow vulnerability, we apply *Cash* to a set of popular network applications that are known to have such a vulnerability. The list of applications and their characteristics are shown in Table 7. At the time of writing this paper, BCC still cannot correctly compile these network applications. because of a BCC bug [4] in the `nss` (name-service switch) library, which is needed by all network applications. Because of this bug, the bounds-checking code BCC generates will cause spurious bounds violations in `nss_parse_service_list`,

Program Name	Lines of Code	Array-Using Loops	>3 Arrays
Qpopper-4.0	32104	67	1 (0.9%)
Apache-1.3.20	51974	355	12 (0.5%)
Sendmail-8.11.3	73612	217	24 (1.4%)
Wu-ftpd-2.6.1	28055	138	1 (0.4%)
Pure-ftpd-1.0.16b	22693	45	1 (0.5%)
Bind-8.3.4	46844	734	22 (0.6%)

Table 7: *Characteristics of a set of popular network applications that are known to have buffer overflow vulnerability. The source code line count includes all the libraries used in the programs, excluding* `libc`.

which is used internally by the GNU C library's name-service switch. Therefore, for network applications, we only compare the results from *Cash* and GCC.

Program Name	Latency Penalty	Throughput Penalty	Space Overhead
Qpopper	6.5%	6.1%	60.1%
Apache	3.3%	3.2%	56.3%
Sendmail	9.8%	8.9%	44.8%
Wu-ftpd	2.5%	2.4%	68.3%
Pure-ftpd	3.3%	3.2%	63.4%
Bind	4.4%	4.3%	53.6%

Table 8: *The latency/throughput penalty and space overhead of each network application compiled under* Cash *when compared with the baseline case without bound checking.*

To evaluate the performance of network applications, we used two client machines (one 300-MHz Pentium-2 with 128MB memory and the other 1.5-GHz Pentium-4 with 256 MB memory), that continuously send 2000 requests to a server machine (1.1-GHZ Pentium-3 with 512 MB memory) over a 100Mbps Ethernet link. The server machine's kernel was modified to record the creation and termination time of each forked process. The throughput of a network application running on the server machine is calculated by dividing 2000 with the time interval between creation of the first forked process and termination of the last forked process. The latency is calculated by taking the average of the CPU time used by the 2000 forked processes. The Apache web server program is handled separately in this study. We configured Apache to handle each incoming request with a single child process so that we could accurately measure the latency of each Web request.

We measured the latency of the most common operation for each of these network applications when the bound checking mechanism in *Cash* is turned on and turned off. The operation measured is sending a mail for Sendmail, retrieving a web page for Apache, getting a file for Wu-ftpd, answering a DNS query for Bind, and retrieving mails for Qpopper. For network applications that can potentially involve disk access, such as Apache, we warmed up the applications with a few runs before taking the 10 measurements used in computing the average. The latency penalty for these applications ranges from 2.5% (Wu-ftpd) to 9.8% (Sendmail), and the throughput penalty ranges from 2.4% (Wu-ftpd) to 8.9% (Sendmail), as shown in Table 8. In general, these numbers are consistent with the results from micro-benchmarking, and demonstrate

that *Cash* is indeed a highly efficient bound checking mechanism that is applicable to a wide variety of applications. The space overhead results in Table 8 are higher than those in Table 2.

A major concern early in the *Cash* project is that the number of segment registers (currently 3) is so small as to cause frequent fall-back to software bound check. Because *Cash* only checks array references within loops, a small number of segment registers is a problem only when the body of a loop uses more than 3 arrays/buffers. That is, the limit on the number of simultaneous array uses is per loop, not per function, or even per program. To isolate the performance cost associated with this problem, we measure the number of loops that involve array references, and the number of loops that involve more than 3 distinct arrays (called *spilled loops*) during the execution of these network applications, and the results are shown in Table 7. The percentage numbers within the parenthesis provide the percentage of loop iterations that are executed in the experiments and that belong to spilled loops. The percentage of static loops in each application that use more than 3 arrays is below 3.5% for all applications except Sendmail, which is at 11%. Unsurprisingly, Sendmail also carries the highest latency and throughput penalty.

4.5 Other Performance Factors

Another potential issue is the number of segments needed in an entire application, because the total number of segments available is 8191. Our results show that the total number of segments used is within 10 segments for the micro-benchmark applications, 163 segments for the macro-benchmark, and 292 for the network applications. Therefore, the budget, 8191, seems more than sufficient for many applications. Another hidden performance cost is increased pointer variable copying overhead due to multi-word pointer representation. Because BCC uses a 3-word representation, its cost in pointer copying is even higher than *Cash*, which uses a 2-word representation. However, for most numerical programs that require array bound checking, pointer assignments are never sufficiently frequent to cause noticeable performance problems.

One major concern is the overhead associated with LDT modification. Among all 18 tested applications, Toast makes the most requests (415,659 calls) to allocate segments. 223,781 of them (or 53.8% hit ratio) can find a matched segment in the 3-entry cache and 191,878 requests actually need to go into the kernel through the `cash_modify_ldt` call gate to modify the LDT. Each call gate invocation takes 253 cycles, which means that it takes 50,464K cycles for the 191,878 calls, and this is relatively insignificant as compared with Toast's total run time (4,727,612K cycles). Therefore, the overhead of the Toast application compiled under *Cash* is still very small (4.6%) though it makes so many segment allocation requests.

5 Conclusion

Although array bound checking is an old problem, it has seen revived interest recently out of concerns on security breaches exploiting array bound violation. Despite its robustness advantage, most real-world programs do not incorporate array bound checking, and the main hurdle is its performance cost. Whereas almost all previous research in this area focused on static analysis techniques to reduce redundant bound checks and thus minimize the checking overhead, this work took a completely different approach that relies on hardware features available in the X86 architecture, which accounts for more than 90% of the worldwide PC market. The main idea of our approach is to organize array reference instructions in such a way that the segment limit check mechanism in the X86 architecture's virtual memory hardware effectively performs array bound check. As a result, the proposed approach, called *Cash*, does not incur any per-array-reference

overhead most of the time, because bound checking is done by the segmentation hardware for free. However, there are per-program overhead, per-array overhead, and per-array-use overhead associated with the *Cash* approach. We have successfully built a *Cash* prototype based on the bound-checking GCC compiler under Red Hat Linux 7.2. The current *Cash* prototype can check bounds for array pointers as well as general pointers. The empirical performance measurements from running a set of numerical kernels that use array references extensively on the *Cash* prototype demonstrate that the *Cash* approach can reduce the array bound checking overhead of a set of popular network applications to under 9.8% compared with the baseline case that does not perform any bound checking.

As for the performance difference between *Cash* and BCC, it has more to do with the fact that it has to perform software array bound checks for references inside loops, and less to do with the fact that it also needs to check references outside the loops. After all, most of the computation time of the test applications is spent on loops. In fact, the micro-benchmark results from Table 1, where the test applications contain only loops, show that BCC actually performs even worse than *Cash* when there are no non-loop array references.

Although the *Cash* approach reduces the array bound checking overhead to an unprecedentedly low level, it relies on a specific hardware feature of the X86 architecture, and thus is not as portable as other software-only approaches. We recognize this limitation. However, we believe that the X86 architecture has a long life time ahead, especially in view of the recent announcement that Intel is planning to develop a 64-bit version of its X86 architecture that will evolve in parallel with its Itanium line.

Acknowledgment

This research is supported by NSF awards SCI-0401777, CNS-0410694 and CNS-0435373 and Rether Networks Inc.

References

[1] Bruce Perens. Electric fence: a malloc() debugger for linux and unix. *http://perens.com/FreeSoftware/*.

[2] Chris Bentley, Scott A. Watterson, and David K. Lowenthal. A comparison of array bounds checking on superscalar and vliw architectures. *submitted to the annual IEEE Workshop on Workload Characterization*, September 2002.

[3] Crispan Cowan, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.

[4] GCC. Bounds-checking gcc. *http://www.gnu.org/software/gcc/projects /bp/main.html*.

[5] Glenn Pearson. Array bounds checking with turbo c. *Dr. Dobb's Journal of Software Tools*, 16(5):72, 74, 78–79, 81–82, 104–107, May 1991.

[6] Harish Patil and Charles N. Fischer. Efficient run-time monitoring using shadow processing. In *Proceedings of Automated and Algorithmic Debugging Workshop*, pages 119–132, 1995.

[7] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, 1998.

[8] Hongwei Xi and Songtao Xia. Towards array bound check elimination in java virtual machine language. In *Proceedings of CASCON '99*, pages 110–125, Mississauga, Ontario, November 1999.

[9] Intel. IA-32 Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference. *http://www.intel.com/design/Pentium4/manuals/*.

[10] Intel. Ia-32 intel architecture software developer's manual. volume 3: System programming guide. *http://developer.intel.com/design/pentium4/manuals/245472.htm*.

[11] J. M. Asuru. Optimization of array subscript range checks. *ACM letters on Programming Languages and Systems*, 1(2):109–118, June 1992.

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

[13] P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–278, 1995.

[14] Manish Prasad and Tzi-cker Chiueh. A binary rewriting approach to stack-based buffer overflow attacks. In *in Proceedings of 2003 USENIX Conference*, June 2003.

[15] Rajiv Gupta. A fresh look at optimizing array bound checking. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 272–282, 1990.

[16] Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March-December 1993.

[17] Rastislav Bodik and Rajiv Gupta and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. *SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–333, 2000.

[18] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compiler time solution to buffer overflow attacks. In *in Proceedings of International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, April 2001.

[19] Tzi-cker Chiueh and Ganesh Venkitachalam and Prashant Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *in Proceedings of 17th ACM Symposium on Operating Systems Principles*, Charleston, SC, December 1999.